Bruce MacLennan

PRINCIPLES OF COMPUTER PROGRAMMING
A Mathematical Approach

Harlan D. Mills

Fellow in the Federal Systems Division of IBM
and Professor of Computer Science at the University of Maryland

Victor R. Basili

Professor and Chairman,
Department of Computer Science
University of Maryland

John D. Gannon

Associate Professor,
Department of Computer Science
and Institute for Advanced Computer Studies
University of Maryland

Richard G. Hamlet

Department of Computer Science and Engineering
Oregon Graduate Center

wcb

Wm. C. Brown Publishers
Dubuque, Iowa

Cover photo credit: The image on the cover is a translucent view of a portion of a recently discovered minimal surface without self-intersections. A minimal surface is one which is locally area-minimizing, somewhat like a soap film spanning a closed wire contour. The full surface is equivalent to a torus from which three points have been removed. It has no edges (in mathematicians parlance, it is complete). Up until 1984, it was thought that such a surface did not exist. The mathematical existence of this example was established by David Hoffman at the University of Massachusetts, Amherst and William Meeks III of Rice University. This work was greatly aided by the use of computer graphics, specifically the work of James T. Hoffman at the University of Massachusetts, who created the image using a graphical programming language, VPL, of which he is the author.

Copyright © 1987 by Allyn and Bacon, Inc.

Copyright © 1988 by Wm. C. Brown Publishers. All rights reserved

Library of Congress Catalog Card Number: 85–23026

ISBN 0–697–06841–2

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Printed in the United States of America by Wm. C. Brown Publishers
2460 Kerper Boulevard, Dubuque, IA 52001

10 9 8 7 6 5

TABLE OF CONTENTS

PREFACE . . . *xvi*

PART I: PROGRAMMING IN CF PASCAL . . . *1*

Programming in a Systematic Way . . . *2*

CF (Character and File) Pascal . . . *3*

Ruler and Compass Constructions . . . *3*

CHAPTER 1 AN INTRODUCTION TO CF PASCAL . . . *7*

1.1 Pascal Programs . . . *7*

1.1.1 Exercises . . . *10*

1.2 Sequential Execution . . . *10*

1.2.1 Exercises . . . *14*

1.3 Character Data . . . *15*

1.3.1 Normal Execution and Aborted Execution . . . *20*

1.3.2 The Assignment Statement . . . *20*

1.3.3 Exercises . . . *22*

1.4 Conditional Execution . . . *23*

1.4.1 Conditions . . . *23*

1.4.2 The IF Statement . . . *25*

1.4.3 The BEGIN Statement . . . *26*

1.4.4 Exercises . . . *29*

1.5 Iterative Execution . . . *30*

1.5.1 The WHILE Statement . . . *30*

1.5.2 Endless Execution . . . *32*

1.5.3 Programs MaxOfAll and CopyOdds . . . *32*

1.5.4 Exercises . . . *35*

1.6 Reading CF Pascal Programs . . . *36*

1.6.1 Program DoubleChars . . . *36*

1.6.2 Program TwoCharacterWords . . . *37*

1.6.3 Program CountChars . . . *40*

1.6.4 Execution-Equivalent Programs . . . *42*

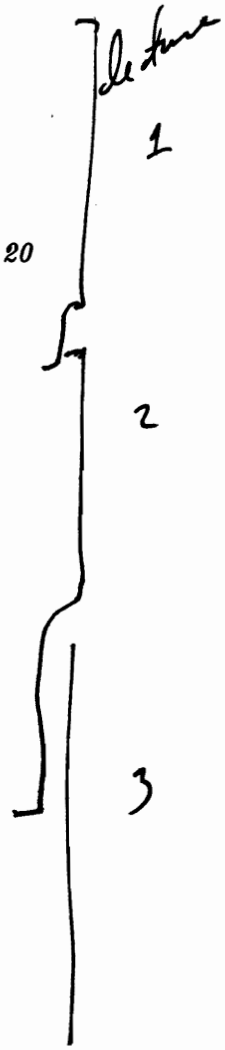
1.6.5 Exercises . . . *43*

1.7 Chapter Summary . . . *44*

CHAPTER 2 WRITING PROGRAMS IN CF PASCAL . . . *45*

2.1 An Introduction to CF Pascal Syntax . . . *45*

2.1.1 Program Syntax . . . *46*



2.1.2 Syntax Trees . . . *48*
2.1.3 Block Syntax . . . *51*
2.1.4 Exercises . . . *53*
2.2 The Syntax of CF Pascal Statements . . . *55*
2.2.1 Exercises . . . *62*
2.3 Program Design and Development . . . *64*
2.3.1 Paul Revere's Ride . . . *65*
2.3.2 Design Parts and Development Programs . . . *68*
2.3.3 Sarah Revere's Letter . . . *69*
2.3.4 Principles of Program Design and Development . . . *76*
2.3.5 Exercises . . . *77*
2.4 Program Organization with Procedures . . . *78*
2.4.1 Simple Procedures . . . *78*
2.4.2 CountChars Again . . . *82*
2.4.3 Reusing Procedures . . . *88*
2.4.4 Exercises . . . *89*
2.5 Chapter Summary . . . *90*

CHAPTER 3 BOOLEAN LOGIC AND CONDITIONAL EXECUTION . . . *92*
3.1 Sorting with Conditional Execution . . . *92*
3.1.1 The IFSort Programs . . . *92*
3.1.2 The MinSort Programs . . . *96*
3.1.3 Exercises . . . *98*
3.2 Boolean Logic . . . *99*
3.2.1 Boolean Operators . . . *100*
3.2.2 Boolean Expressions . . . *102*
3.2.3 Boolean Identities . . . *103*
3.2.4 Exercises . . . *104*
3.3 Boolean Conditions in CF Pascal . . . *105*
3.3.1 The MinSort Programs Again . . . *107*
3.3.2 Exercises . . . *108*
3.4 Chapter Summary . . . *109*

CHAPTER 4 TEXT FILES AND ITERATIVE EXECUTION . . . *110*
4.1 TEXT Files . . . *110*
4.1.1 File Operations . . . *110*
4.1.2 File Syntax and Semantics . . . *111*
4.1.3 Copying Files . . . *111*
4.1.4 Splitting Files . . . *114*

4.1.5 Exercises . . . *116*
4.2 Sorting with Iterative Execution . . . *116*
4.2.1 Program SelectSort . . . *116*
4.2.2 Structural Testing . . . *126*
4.2.3 Analysis of SelectSort . . . *127*
4.2.4 Exercises . . . *129*
4.3 TEXT File Markers . . . *130*
4.3.1 End of Line Markers . . . *130*
4.3.2 End of File . . . *131*
4.3.3 Copying Lines . . . *131*
4.3.4 BubbleSort . . . *133*
4.3.5 Exercises . . . *135*
4.4 Chapter Summary . . . *136*

PART II: PROGRAM CALCULUS OF CF PASCAL . . . *138*
The Next Step . . . *138*
What is a Program Calculus? . . . *138*
The Mathematical Basis for the Program Calculus . . . *138*
Programs as Mathematical Objects . . . *139*
The Syntax and Semantics of CF Pascal . . . *140*

CHAPTER 5 THE SEMANTICS OF CF PASCAL . . . *142*
5.1 Character Strings . . . *142*
5.1.1 String Concatenation . . . *144*
5.1.2 Substrings . . . *145*
5.1.3 String Composition . . . *145*
5.1.4 String Decomposition . . . *146*
5.1.5 Exercises . . . *147*
5.2 List Structures . . . *148*
5.2.1 List Operations . . . *149*
5.2.2 List Descriptions of Files . . . *150*
5.2.3 Exercises . . . *152*
5.3 Set Structures . . . *153*
5.3.1 Rules for Describing Sets . . . *154*
5.3.2 Set Operations . . . *155*
5.3.3 Relations and Functions . . . *156*
5.3.4 Exercises . . . *158*
5.4 Program Meanings . . . *159*
5.4.1 The Box Notation . . . *159*

5.4.2 Determining Program Meanings Directly . . . *160*
5.4.3 Exercises . . . *162*
5.5 Chapter Summary . . . *162*

CHAPTER 6 THE CALCULUS OF SEQUENTIAL PROGRAMS . . . *163*
6.1 The Meanings of Program Parts . . . *163*
6.1.1 Execution States . . . *163*
6.1.2 The Meaning of the Program Header and Period . . . *165*
6.1.3 Compositions of Relations and Functions . . . *166*
6.1.4 The Meaning of Declarations . . . *168*
6.1.5 The Meaning of Blocks . . . *170*
6.1.6 Exercises . . . *172*
6.2 The Meaning of Sequential Statements . . . *173*
6.2.1 Assignment Statements . . . *173*
6.2.2 Null Statements . . . *175*
6.2.3 BEGIN Statements . . . *175*
6.2.4 WRITE Statements . . . *176*
6.2.5 READ Statements . . . *177*
6.2.6 Simple Procedure Statements . . . *179*
6.2.7 Exercises . . . *179*
6.3 The Analysis of Sequential Statements . . . *180*
6.3.1 Concurrent Assignments . . . *180*
6.3.2 Symbolic Execution and Trace Tables . . . *182*
6.3.3 Nested BEGIN Statements . . . *184*
6.3.4 Exercises . . . *185*
6.4 The Design of Sequential Statements . . . *186*
6.4.1 A Design Rule for BEGIN statements . . . *187*
6.4.2 Exercises . . . *190*
6.5 Chapter Summary . . . *191*

CHAPTER 7 THE CALCULUS OF CONDITIONAL STATEMENTS . . . *192*
7.1 The Meaning of Conditional Statements . . . *192*
7.1.1 Meaning of Boolean Expressions . . . *192*
7.1.2 Meaning of IF Statements . . . *195*
7.1.3 Exercises . . . *196*
7.2 The Analysis of Conditional Statements . . . *197*
7.2.1 Conditional Assignments . . . *197*
7.2.2 Conditional Trace Tables . . . *200*
7.2.3 Comparing Conditional Assignments . . . *204*

7.2.4 Exercises . . . *207*
7.3 The Design of Conditional Statements . . . *210*
7.3.1 Design Rule for IF statements . . . *210*
7.3.2 Designing IF Statements for Conditional Assignments . . . *212*
7.3.3 Exercises . . . *213*
7.4 Chapter Summary . . . *213*

CHAPTER 8 THE CALCULUS OF ITERATION STATEMENTS . . . *214*
8.1 The Meaning of Iteration Statements . . . *214*
8.1.1 The Meaning of WHILE Statements Defined Recursively . . . *214*
8.1.2 Counting Iterations in WHILE Statements . . . *216*
8.1.3 Exercises . . . *219*
8.2 The Analysis of Iteration Statements . . . *220*
8.2.1 The WHILE Statement Verification Rule . . . *220*
8.2.2 Composition of Conditional Assignments . . . *222*
8.2.3 WHILE Statement Verification Examples . . . *223*
8.2.4 Exercises . . . *229*
8.3 The Design of Iteration Statements . . . *229*
8.3.1 WHILE Statement Design Rules . . . *230*
8.3.2 WHILE Statement Design from Concurrent Assignments . . . *233*
8.3.3 Exercises . . . *233*
8.4 The Theory of Iteration Statements . . . *234*
8.4.1 A Proof By Contradiction . . . *234*
8.4.2 The WHILE Statement Verification Rule . . . *234*
8.4.3 The Undecidability of WHILE Statement Termination . . . *236*
8.4.4 Exercises . . . *238*
8.5 Chapter Summary . . . *238*

PART III: DEVELOPING LARGER PROGRAMS . . . *239*
Procedures and Libraries . . . *239*
Recursion . . . *239*
Specification and Correctness . . . *239*

CHAPTER 9 CF PASCAL PROCEDURES . . . *241*
9.1 Procedures with Parameters . . . *241*
9.1.1 Simple Procedures and Parameters . . . *241*
9.1.2 Procedure Parameter Syntax and Context Rules . . . *243*
9.1.3 Splitting Files . . . *243*
9.1.4 The Calculus of Parameters . . . *245*

9.1.5 Exercises . . . *246*
9.2 Procedures with Declarations . . . *249*
9.2.1 Identically Named Variables . . . *251*
9.2.2 Scope of Identifiers . . . *252*
9.2.3 Syntax and Context Rules . . . *253*
9.2.4 String Manipulation Procedures . . . *254*
9.2.5 Identifier Confusion . . . *256*
9.2.6 Exercises . . . *259*
9.3 Processing Natural Numbers . . . *261*
9.3.1 The Natural Numbers . . . *261*
9.3.2 One-Digit Addition . . . *262*
9.3.3 Representation and Comparison of Natural Numbers . . . *264*
9.3.4 Addition of Natural Numbers . . . *264*
9.3.5 Multiplication of Natural Numbers . . . *267*
9.3.6 A Starter Library for Natural Numbers . . . *269*
9.3.7 Exercises . . . *270*
9.4 Chapter Summary . . . *272*

CHAPTER 10 PROGRAMMING WITH WORD STRINGS . . . *273*
10.1 Word Strings . . . *273*
10.1.1 Words and Languages . . . *273*
10.1.2 Using BNF in Program Design . . . *273*
10.1.3 Removing Extra Blanks . . . *274*
10.1.4 Testing RemoveExtraBlanks . . . *277*
10.1.5 Analyzing RemoveExtraBlanks . . . *279*
10.1.6 Exercises . . . *291*
10.2 Sorting Word Strings . . . *292*
10.2.1 Sorting by Splitting and Merging . . . *293*
10.2.2 Merging Runs . . . *295*
10.2.3 Splitting Runs . . . *297*
10.2.4 Completing MergeSort1 . . . *299*
10.2.5 Exercises . . . *300*
10.3 An Improved MergeSort . . . *301*
10.3.1 Exercises . . . *306*
10.4 Chapter Summary . . . *307*

CHAPTER 11 PROGRAMMING WITH RECURSION . . . *308*
11.1 Recursive Procedures in CF Pascal . . . *308*
11.1.1 Problem Solving with Recursion . . . *308*

11.1.2 Recursive Procedure Execution . . . *309*
11.1.3 Exercises . . . *312*
11.2 Sorting and Reversing with Recursion . . . *313*
11.2.1 Recursive Sort . . . *313*
11.2.2 Recursive Reverse . . . *317*
11.2.3 Exercises . . . *318*
11.3 Solving Problems with Recursion . . . *319*
11.3.1 The Towers of Hanoi . . . *319*
11.3.2 Mathematical Induction . . . *321*
11.3.3 A Recursive Procedure for the Towers of Hanoi . . . *323*
11.3.4 Exercises . . . *324*
11.4 The Calculus of Recursive Procedures . . . *325*
11.4.1 Calculating the Meaning of Recursive Procedure Calls . . . *325*
11.4.2 A Recursive Procedure Verification Rule . . . *326*
11.4.3 Examples of Recursive Procedure Verification . . . *328*
11.4.4 Proof of the Recursive Procedure Verification Rule . . . *331*
11.4.5 Exercises . . . *332*
11.5 Chapter Summary . . . *332*

CHAPTER 12 SYSTEMATIC PROGRAM DESIGN AND CORRECTNESS . . . *333*
12.1 Program Specifications . . . *333*
12.1.1 A Specification for RemoveExtraBlanks . . . *334*
12.1.2 Exercises . . . *335*
12.2 Program Correctness . . . *335*
12.2.1 Mathematical Formality and Proof . . . *337*
12.2.2 Mathematical Proofs . . . *337*
12.2.3 Exercises . . . *338*
12.3 Program Design . . . *339*
12.3.1 Correctness of Program Parts . . . *339*
12.3.2 Conditional Assignments as Specifications . . . *339*
12.3.3 Program Design by Function Refinement . . . *340*
12.3.4 Exercises . . . *341*
12.4 Chapter Summary . . . *341*

PART IV: PROGRAM DESIGN IN D PASCAL . . . *342*
Data Abstractions . . . *342*
D (Design) Pascal . . . *343*

CHAPTER 13 PROGRAM MODULES . . . *344*

13.1 Numbers . . . *345*
13.1.1 Exercises . . . *348*
13.2 Data Abstractions . . . *349*
13.2.1 Exercises . . . *351*
13.3 State Machines . . . *351*
13.3.1 Exercises . . . *354*
13.4 Queue Modules . . . *354*
13.4.1 SlowQueue . . . *355*
13.4.2 FastQueue . . . *357*
13.4.3 RemoveExtraBlanks . . . *366*
13.4.4 Exercises . . . *369*
13.5 Other Storage and Retrieval Structures . . . *369*
13.5.1 Exercises . . . *370*
13.6 Chapter Summary . . . *371*

CHAPTER 14 ORDINAL DATA TYPES . . . *372*
14.1 Enumerated Types . . . *373*
14.1.1 Syntax and Meaning of Enumerated Type Expressions . . . *375*
14.1.2 Enumerated Type Input/Output . . . *375*
14.1.3 Comparing Dates . . . *377*
14.1.4 Exercises . . . *378*
14.2 Type Boolean . . . *379*
14.2.1 Boolean Syntax . . . *380*
14.2.2 The Meaning of Boolean Expressions . . . *383*
14.2.3 Boolean Input/Output . . . *384*
14.2.4 Searching a File . . . *384*
14.2.5 Exercises . . . *385*
14.3 Type Integer . . . *386*
14.3.1 Counting Blanks . . . *387*
14.3.2 Integer Input/Output . . . *389*
14.3.3 Overflow . . . *392*
14.3.4 Integer Syntax . . . *396*
14.3.5 Meaning of Integer Expressions . . . *396*
14.3.6 Converting Characters to Numbers . . . *397*
14.3.7 Exercises . . . *399*
14.4 Subrange Types . . . *400*
14.4.1 Subrange Assignments . . . *401*
14.4.2 Temperature Conversion . . . *401*
14.4.3 Exercises . . . *402*

14.5 Design and Analysis with Ordinal Types . . . *403*
14.5.1 Design Rules . . . *403*
14.5.2 Program Analysis . . . *404*
14.5.3 Exercises . . . *408*
14.6 Summary . . . *411*

CHAPTER 15 THE PROGRAM STRUCTURE OF D PASCAL . . . *413*
15.1 External Files . . . *413*
15.1.1 Exercises . . . *415*
15.2 Constants . . . *415*
15.2.1 Exercises . . . *416*
15.3 Types . . . *417*
15.3.1 Exercises . . . *419*
15.4 Value and Variable Parameters . . . *420*
15.4.1 Exercises . . . *423*
15.5 Functions . . . *423*
15.5.1 Side Effects in Functions . . . *425*
15.5.2 Predefined Functions . . . *426*
15.5.3 Exercises . . . *430*
15.6 Forward Declarations . . . *433*
15.7 Summary . . . *434*

CHAPTER 16 AGGREGATE DATA TYPES IN D PASCAL . . . *435*
16.1 Sets . . . *435*
16.1.1 Counting Characters . . . *437*
16.1.2 Rules of Syntax and Meaning . . . *438*
16.1.3 Prime Numbers . . . *439*
16.1.4 Exercises . . . *441*
16.2 Files . . . *442*
16.2.1 Non-TEXT Files . . . *443*
16.2.2 OddBeforeEven Program System . . . *445*
16.2.3 TEXT Files . . . *448*
16.2.4 Justifying Text with InsertExtraBlanks . . . *449*
16.2.5 Interactive Input/Output . . . *454*
16.2.6 Exercises . . . *456*
16.3 Records . . . *457*
16.3.1 Record Syntax and Meaning . . . *459*
16.3.2 Sorting Dates . . . *460*
16.3.3 Exercises . . . *464*

16.4 WITH Statement . . . *465*
16.4.1 Exercises . . . *466*
16.5 Summary . . . *466*

CHAPTER 17 DESIGNING SEQUENTIAL CONTROL . . . *467*
17.1 The CASE Statement . . . *467*
17.1.1 CASE Statement Examples . . . *468*
17.1.2 CASE Statement Definition . . . *470*
17.1.3 Exercises . . . *470*
17.2 The FOR Statement . . . *471*
17.2.1 Examples of FOR Statements . . . *472*
17.2.2 FOR Statement Definition . . . *473*
17.2.3 Block Letters . . . *475*
17.2.4 Analysis of FOR Statements . . . *477*
17.2.5 Exercises . . . *478*
17.3 The REPEAT Statement . . . *480*
17.3.1 REPEAT Statement Definition . . . *480*
17.3.2 Exercises . . . *481*
17.4 Chapter Summary . . . *482*

CHAPTER 18 ABSTRACT DATA TYPES . . . *483*
18.1 Concrete and Abstract Worlds . . . *484*
18.2 Rational Numbers . . . *486*
18.2.1 Representing Rationals . . . *487*
18.2.2 Operations on Rationals . . . *487*
18.2.3 Harmonic Series . . . *491*
18.2.4 An Alternate Representation for Rationals . . . *492*
18.2.5 Re-implemented Operations for Rationals . . . *495*
18.2.6 Exercises . . . *497*
18.3 The Calculus of Abstract Data Types . . . *498*
18.3.1 Verifying an Implementation . . . *500*
18.3.2 Sample Verification . . . *504*
18.3.3 Verifying a Module Use . . . *507*
18.3.4 Exercises . . . *508*
18.4 Chapter Summary . . . *511*

PART V: PROGRAM DESIGN IN O PASCAL . . . *512*

CHAPTER 19 AGGREGATE DATA TYPES IN O PASCAL . . . *513*

19.1 Arrays . . . *513*
19.1.1 Syntax and Semantics for Arrays . . . *517*
19.1.2 Array parameters . . . *519*
19.1.3 Exercises . . . *520*
19.2 Implementing Abstract Types with Arrays . . . *521*
19.2.1 Finite TEXT Files . . . *521*
19.2.2 Finite Stacks . . . *523*
19.2.3 Exercises . . . *524*
19.3 Searching and Sorting Arrays . . . *525*
19.3.1 Searching an Array . . . *526*
19.3.2 Sorting Arrays . . . *528*
19.3.3 Exercises . . . *532*
19.4 Array Extensions and Applications . . . *533*
19.4.1 The PACKED Attribute . . . *533*
19.4.2 Multidimensional Arrays . . . *538*
19.4.3 Exercises . . . *541*
19.5 Analyzing Programs with Arrays . . . *542*
19.5.1 Exchange . . . *543*
19.5.2 A WHILE Statement Example . . . *544*
19.5.3 Exercises . . . *546*
19.6 Chapter Summary . . . *547*

CHAPTER 20 LINKED STRUCTURES . . . *548*
20.1 Implementing Linked Structures With Arrays . . . *549*
20.1.1 A Simple Linked List . . . *549*
20.1.2 Insertion Sort . . . *552*
20.1.3 Evaluating Arithmetic Expressions . . . *555*
20.1.4 Tree Structures . . . *557*
20.1.5 Exercises . . . *558*
20.2 Pointers . . . *559*
20.2.1 Pointer and Target Values in the Execution State . . . *561*
20.2.2 Implementing Linked Structures with Pointers . . . *563*
20.2.3 Insertion Sort . . . *566*
20.2.4 A Binary Tree Sort . . . *567*
20.2.5 Pointers, Functions, and Abstract Types . . . *569*
20.2.6 Exercises . . . *571*
20.3 Chapter Summary . . . *572*

CHAPTER 21 GOTO STATEMENTS AND STRUCTURED PROGRAMS . . . *573*

21.1 The GOTO Statement . . . *573*
21.1.1 IF and WHILE Statements with GOTO . . . *574*
21.1.2 Escaping from an Error Condition . . . *574*
21.1.3 Transfers into Composite Statements . . . *579*
21.1.4 Exercises . . . *579*
21.2 Analysis of Programs with GOTO Statements . . . *580*
21.2.1 Structuring an Unstructured Program . . . *581*
21.2.2 A Structuring Example . . . *583*
21.2.3 Exercises . . . *589*
21.3 Chapter Summary . . . *592*

CHAPTER 22 REAL NUMBERS . . . *593*
22.1 Type REAL . . . *593*
22.1.1 Syntax and Operations . . . *593*
22.1.2 Real Input/Output . . . *595*
22.1.3 Compound Interest and Simulation . . . *596*
22.1.4 Exercises . . . *605*
22.2 Roundoff Errors . . . *606*
22.2.1 Roundoff Notation . . . *609*
22.2.2 An Analysis of Roundoff Error . . . *609*
22.2.3 Exercises . . . *610*
22.3 Chapter Summary . . . *612*

CHAPTER 23 PLANNING FOR CHANGE . . . *613*
23.1 CONST and TYPE Declarations . . . *613*
23.1.1 Exercises . . . *616*
23.2 PROCEDURE and FUNCTION Parameters . . . *616*
23.2.1 Syntax for Parameters . . . *617*
23.2.2 Point of Evaluation . . . *618*
23.2.3 Nonlocal References . . . *619*
23.2.4 Zero Finding using Bisection . . . *620*
23.2.5 Sorting Different Kinds of Data . . . *622*
23.2.6 Exercises . . . *625*
23.3 Variant Records . . . *627*
23.3.1 Variant Record Syntax . . . *630*
23.3.2 Pointers to Variant Records . . . *630*
23.3.3 Concordance . . . *631*
23.3.4 Exercises . . . *637*
23.4 Chapter Summary . . . *638*

CHAPTER 24 ONE LAST SOLUTION . . . *639*
24.1 The Prime Number Problem Again . . . *639*
24.1.1 Developing and Proving the Solution . . . *640*
24.1.2 Verifying the Data Abstraction . . . *647*
24.1.3 Integrating and Testing the Design Parts . . . *651*
24.2 Chapter Summary . . . *655*

Appendix I: Collected Syntax for CF Pascal . . . *656*
Syntax Rules . . . *656*
Context Rules . . . *657*

Appendix II: STANDARD PASCAL WORDS . . . *659*
Reserved Words . . . *659*
Standard Identifiers . . . *659*

Appendix III: Syntax of Pascal . . . *660*

INDEX . . . *665*

PREFACE

In any human activity we progress in understanding by reducing complexity to simplicity. In programming methodology, the story is quite familiar. As a result of work in structured programming by Dijkstra, Hoare, Parnas, Gries, Wirth, and many others, we have systematic procedures for program design. As a result of work in functional and denotational semantics by Turing, Kleene, Scott, and others, we have systematic procedures for proving program correctness. As a result, we now understand many deep simplicities that were formerly shrouded in the mystery and complexity of programming lore.

However, since these understandings are quite recent, people regard them as graduate subjects. Our belief is that the time has come for computer science students to start out with these ideas, not end up with them. At their roots, these deep simplicities can be formulated in classical mathematics, discrete mathematics. This is material that undergraduates can learn easily, because understanding it does not require a wide context of programming experience. We believe that the world will be astonished at how proficient young people can become in programming if we start them off right and expect great things from them!

Our objective is a basic course for computer science, much as calculus is a basic course for mathematics and physical sciences, concerned primarily with methodology rather than subject matter. In fact we introduce a "program calculus" in this book, which deals with deriving functions from programs. The functions in the program calculus are the input-to-output functions defined by the linear text of the program, itself a discrete, character-valued function.

This book will require more mathematical maturity than typical textbooks in programming today. The mathematics *per se* is not difficult; the difficulty will occur in learning to use mathematical reasoning in programming. It is natural to approach programming as a puzzle-solving activity, in which solutions are found by cleverness and persistence in debugging, rather than as a mathematically based problem-solving activity. As analogies, it is natural to look at the keys while typing and to dog paddle while swimming. Touch typing and free-style swimming are unnatural activities that take faith and practice to master (and as much faith as practice!). Structured programming based on the program calculus and mathematical correctness is an unnatural activity that also takes faith and practice to master. But the improvement of structured programming over what comes naturally is as dramatic as that of free style over dog paddling, as touch typing over hunt-and-peck.

To minimize the struggle programmers have in finding bugs and getting their programs to run, we intend them to keep their programming under better intellectual control and make far fewer errors. We understand the need in human learning and problem solving for unstructured thinking and experimentation, but we believe it can be isolated and accelerated to a great extent in programming with better methodology. In fact, we intend to replace the anxiety of haphazard success in programming with the joy of intellectual control, consistent success, and predictability in programming.

To the Teacher

Drafts of this text have been used for several years in the two-semester, eight-credit introductory course for computer science majors at the University of Maryland. The course consists of two and one half hours of lecture and two hours of recitation each week. The lectures covered the text book and the recitation covered the computer system, the idiosyncrasies of the local Pascal compiler, extra examples, and reviews of solutions to the homeworks and quizzes. The course has been taught by each of the authors and by other instructors at the University of Maryland.

The course requires new ways of thinking even for experienced computer science educators. The material is deep and quite different from the traditional "first course" in programming. However, the authors believe the material is no harder for students than traditional introductory courses in other scientific disciplines such as calculus or physics. The authors have a strong sense of satisfaction teaching this course because it formalizes basic ideas of computer science and provides the student with a solid foundation in the principles of programming.

Because many students learn about computers before they come to the university, they enter with very different computer backgrounds. Those who know a lot about programming, even in Pascal, will not be able to skip this course. It is *not* a course in Pascal programming. The mix of backgrounds in computers and in mathematics causes student expectations and reactions to vary widely. Some are disappointed that they are not writing big, realistic programs. They may think the material is too simple until they are surprised by the real depth in the course. Others are initially overwhelmed because of their lack of background or intimidated by the programming knowledge of classmates. There will always be a few good students who have trouble simply because the course does not live up to their expectations. It is more important than usual to make it clear early what the course is about and what is expected. This motivation should be reinforced at regular intervals.

There is a great deal of material covered in this book so a teacher may pick and choose among the topics and levels of depth. Certain chapters and sections are critical because they represent the core ideas of the book and the material in later chapters builds upon them. For example, Chapter 2 covers the syntactic correctness of programs using context-free grammars. Grammars are later used to discuss functional testing, program specifications, and semantic correctness. Chapter 5 is the basis of the mathematical formulation of the programming language semantics and is used throughout the book for correctness and specification. On the other hand, in many sections, there are more examples given than might be covered in the lecture section. These extra examples can be covered in the recitation sections or read independently by the student who needs to see several examples. Certain sections can be omitted. For example, in Chapter 10, sections 2, 3, and 4 can be omitted without penalty because they take basic ideas and use them to build larger examples.

A sample lecture schedule for two fifteen-week semesters with a midterm exam is given below. It shows the time that might be given to each section, what should be emphasized, and what could be left out.

Week	Sections Covered (* emphasis)	
	First Semester	Second Semester
1	1.1-1.4	Review of program calculus (Part II)
2	1.5, 1.6, 2.1*	13.1-13.5*
3	2.1(cont), 2.2*, 2.3, 2.4	14.1-14.3
4	3.1-3.3	14.3(cont), 14.4, 14.5
5	4.1, 4.2	15.1-15.6
6	5.1-5.3	16.1-16.4
7	5.4*, exam	17.1-17.3, exam
8	6.1, 6.2*, 6.3	18.1-18.3
9	7.1, 7.2*, 7.3	19.1-19.6
10	8.1, 8.2*	20.1, 20.2
11	8.3, 8.4, 9.1	21.1, 21.2
12	9.2*, 9.3	22.1-22.5
13	10.1	23.1-23.3
14	11.1*, 11.2, 11.3	24
15	11.4, 12	

Even though only one language is taught in the course, we believe the student is prepared to learn other languages. In our curriculum at Maryland, students are exposed to many other languages in succeeding courses, which we believe are easier to learn because of the material covered in this course.

Over the years, we have learned that students have more trouble with some topics than with others. Context-free grammars represent a new idea and need to be covered in great detail. The material on sets, lists, functions, and relations needs lots of attention. Even though many students have seen the concepts before, they are not always comfortable with the mathematics or its application, e.g., inputs viewed as a list, execution states viewed as relations. The design rules are not easy to use at first and need practice.

We hope that you find this text as interesting to teach from as we found it to write. We learned a great deal and are still learning, and we wish you the same experience.

Acknowledgements

This book is a product of the Computer Science Department of the University of Maryland, where all the authors have served on the faculty. The ideas have migrated from research topics to graduate courses to undergraduate courses by various individual and joint efforts among the authors. The authors appreciate the opportunities for teaching these courses, and the interest and tolerance of fellow faculty members in developing the book. The authors also appreciate the patience and forbearance of dozens of teachers, teaching assistants, and thousands of students who have struggled with previous versions of the book; many of these teachers, assistants, and students have provided useful suggestions, notably Stanley Dunn, David Elstner, John Havlichek, Rochelle Heller, and David Preston.

Four faculty members at the University of Maryland played a special role in the conception and development of this book. Richard H. Austing was involved in original planning and we appreciate his advice and counsel in both computer science and educational matters. Ben Shneiderman was an advocate of the "spiral" approach the book takes, and made significant suggestions to improve its style overall. John Kohl taught from early drafts, and helped us to appreciate the difficulties students have with new ideas. Harold Reiter taught from later drafts and improved the mathematical consistency.

In June 1982, the Eastern Computer Institute sponsored a one-week course "Teaching a Programming Calculus" for computer science teachers. The Wang Institute hosted the course and provided a significant grant to support scholarships. Nancy Martin at Wang organized and co-taught the course with authors Basili and Mills, providing a comprehensive review and critique of the text.

Lenard B. Pedowitz and Coni Loving supervised the production of earlier drafts using an IBM personal computer.

H. D. M.
V. R. B.
J. D. G.
College Park, MD

R. G. H.
Portland, OR

April, 1986

PART I: PROGRAMMING IN CF PASCAL

Today, everyone is touched by the computer. It has come to affect almost everything in our daily lives from our education to our retirement plans, how we are tested and diagnosed, how we buy our groceries and entertain ourselves, how we are billed. Computers are an important component in our appliances, digital watches, cars, and spacecraft.

Why has this one aspect of technology had such an impact and created an industry that commands over 3 percent of the gross national product ($300 a year for every man, woman, and child in the United States in 1982)? Is it because the computer simplifies our tasks, makes life easier, or increases productivity by doing tedious work quickly? Is it because the information explosion in many areas of science and industry requires more effective processing? Does the computer solve existing problems or provide new opportunities? Computer science and technology do both; they serve present needs and stimulate further progress.

What is a computer? A computer is a machine that can accept, store, and manipulate data. It is not like most machines, which must be guided each time to perform a task. Computers in a sense guide themselves by executing a set of step-by-step instructions called a *program*. To direct a computer to solve a particular problem, someone must write a program, causing the computer to solve that problem. The ability to write another program and change the calculation is what differentiates the computer from the hand calculator. In a hand calculator, hitting the plus button automatically adds the number displayed to the current sum. But to add two numbers the button must be hit twice. A computer can be programmed to perform a lengthy series of operations, and when started will carry them out in sequence to solve a complex problem.

Like a calculator, which accepts a number keyed into it digit by digit so that an arithmetic operation can take place, a computer can accept data from an external source. The data accepted by a computer are called *input data*. A computer can produce data as a result of its operations; for example, the results of a series of computations can be printed. The data produced by a computer are called *output data,* or the solution to the problem.

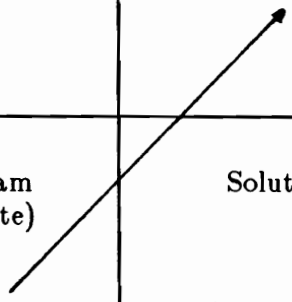
How does a program (a list of instructions) get placed into the computer? First, the program must be written in a form acceptable to the computer—in a *programming language*. The computer must already contain the instructions to read and to recognize the instructions of the programming language as input. The program that reads the instructions in the programming language and prepares them for execution is called a *compiler*. This book is based on Pascal, a programming language invented by Niklaus Wirth. A Pascal compiler translates a set of instructions written in the Pascal language into a *machine program* for the computer. That is, executing a Pascal program typically requires a two-stage process (see Figure 1). In Stage 1, the Pascal program is the input, and the machine program (ready to execute) is the output. In Stage 2 the machine program (just created) transforms problem input data into the solution output data.

Programming is a problem-solving activity. But before solving any problem, you must clearly define the problem and know that a computer can be used to solve it. Next, you must invent a method for finding the problem's solution, and state the method as a set of instructions to follow. The next step is to write these instructions in a language the computer has been programmed to accept—in our case, Pascal. You might ask, if you can solve the problem before writing the instructions for a computer, why do you need the computer? The answer is that the sheer quantity of data or the complexity of the data manipulation makes carrying out the solution tedious or impossible for a human. If either of these premises is false, then you probably do *not* need a computer.

* cf. programmable calculators, which repeat a sequence of operations.

Figure 1: A Two-Stage Process for Executing Pascal Programs

	Stage 1	Stage 2
INPUT	Pascal Program (to be executed)	Problem Data
PROGRAM	Pascal Compiler (in computer)	Machine Program (just created)
OUTPUT	Machine Program (ready to execute)	Solution Data



Programming in a Systematic Way

On the first day in a typing class, you could be shown how to insert a sheet of paper into a typewriter, told how the keys work, and given a short letter to type. You could then spend the rest of the day typing the letter, by the hunt-and-peck method. Chances are, you would start over several times, forgetting to set the margins one time, running words off the edge of the page another, not leaving enough space between the lines, and so on. But by the end of the day you would have an acceptable looking letter, a reasonable facsimile of the original letter that you were given to copy. You could repeat the process the next day with a slightly longer letter, and by the end of the week you would be getting a two- or three-page letter out in a day. It would look like you were doing something useful, just like a professional secretary. You couldn't type before, but you can type now. You have immediate confirmation in your new ability to do useful things.

Unfortunately, we know that learning to type proficiently can't be done that way. A year from now it will still take you a day to get out a five-page memo, but you would be expected to do it in an hour, not a day. What went wrong? You did what came naturally, and you reached a certain level of ability. But you also developed a set of bad habits that will prevent you from ever getting any better. What went wrong was that you learned things that looked useful rather than learning useful things. The deep lesson we learn in typing is not to do something that looks useful the first day but to begin to learn useful skills instead. We know that we should learn to type in a systematic way (that is, without looking at the keys) because it is a better way to type in the long run, even though it is a terrible way to try to type on the first day.

Typing is simple; computer programming is complex. A day of learning in typing is more like several weeks of learning in computer programming. But, just as you can learn to type a letter in a day, you can learn to write a computer program in a week. If someone explains how the computer works, using trial and error, you can get a simple enough program written and run just by doing what comes naturally.

With this book, you can learn how to program computers in a systematic way. You will not be asked to write a program that looks useful to begin with, even though you might be able to. Trying to address adult problems in childish ways leads to bad habits that should never be formed.

In the beginning, you will find that you can get more computing done with a hand calculator than you can with a computer. In time you will acquire a systematic ability to design and write computer programs of far greater capability and complexity than you could construct by doing what comes naturally. So hang on to your hand calculator for the computing you may want to do for now. Don't get trapped by trying to puzzle out by trial and error how to get a computer to do useful things for you immediately—instead, learn the useful skills in the first part of this book. Otherwise, you are back in typing class getting a letter out on the first day.

CF (Character and File) Pascal

Part I introduces programming as a problem-solving activity in CF Pascal, a subset of the Pascal programming language. Later, we will augment CF Pascal into the full Pascal language. In CF Pascal, there is but a single kind of data (namely, characters **A**, **a**, **B**, **b**, ..., **1**, **2**, **3**, ..., **#**, **$**, ... etc.) and a single data structure (namely, character files—sequences of characters that can only be accessed one after the other without backing up). By restricting attention to CF Pascal, we can concentrate on programming instead of the details of programming languages.

CF Pascal is a simple language, but that is not the most important reason why we begin with it. Characters and files are fundamental building blocks for computing. Simple and easily stated problems in programming require thought and organization to solve with characters and files. Yet programming in CF Pascal is a complete microcosm of programming in any language. There are no new problems encountered when programming in full Pascal; the more powerful language features only allow the same problems to be handled more efficiently by a computer.

In the early days of automatic computing, it wasn't obvious that simple characters and sequences of characters would be fundamental to all computing. The earliest electronic computers were inspired by problems of mathematical computation and dealt entirely with numerical data. The entire data processing industry has grown out of the realization that computers can deal with character data as well. But another, far deeper, realization came from experience with computers; it was seen that mathematical computation itself is character processing—no more, no less!

Computers obey the laws of physics and electronics but not the laws of arithmetic. The arithmetic in computers comes from clever design by engineers who have arranged for electronic devices to simulate the arithmetic operations that the computer users desire. In fact, the computer reads and writes only character sequences as it interfaces with humans. The digit "7" and the number seven to humans are the same character **7** to the computer. The restriction to characters in CF Pascal does not eliminate numerical operations; it simply makes the basis for numerical operations more explicit. The most important data organization in computers is in sequences. Data in computers are transferred from one place to another in sequential form. Communication between one computer and another computer or terminal is in sequential form. CF Pascal files access characters in sequence and give us the opportunity to study the sequence as a data structure in a concentrated way.

By concentrating on two fundamental computer ideas of characters and sequences in CF Pascal, it will be possible to illustrate and explain the important concepts of programming in the simplest setting. Later, after these concepts are well understood, it will be clear how to apply them to the more extensive data and structures of full Pascal.

Ruler and Compass Constructions

In mathematics, learning about geometrical constructions using only ruler and compass is a proven educational strategy. By restricting the constructions to simple tools, we concentrate on the ideas, not the apparatus. CF Pascal is intended to be a "ruler and compass" programming language. Just as a good draftsperson soon augments basic tools with protractors, marked rulers, and other labor-savers, so will CF Pascal soon grow to full Pascal. But we can learn more about principles with less complex tools.

Consider the following geometrical problem: given a triangle, construct a rectangle of equal area, using only a ruler and a compass. In thinking about the problem, the idea immediately comes to mind that the area of a triangle is half its base times its altitude. But try telling that to a ruler

and compass! Instead, we will make use of that formula ourselves in deciding on a proper construction. After some more thinking, we arrive at a three-step solution (illustrated in Figure 2):

1. Given a triangle, select any vertex and construct a perpendicular to the line that includes the opposite side.
2. Bisect the perpendicular constructed in step 1.
3. Construct a rectangle with a base equal to the length of the opposite side selected in step 1 and with a height equal to the length of a bisected half of the perpendicular constructed in step 2.

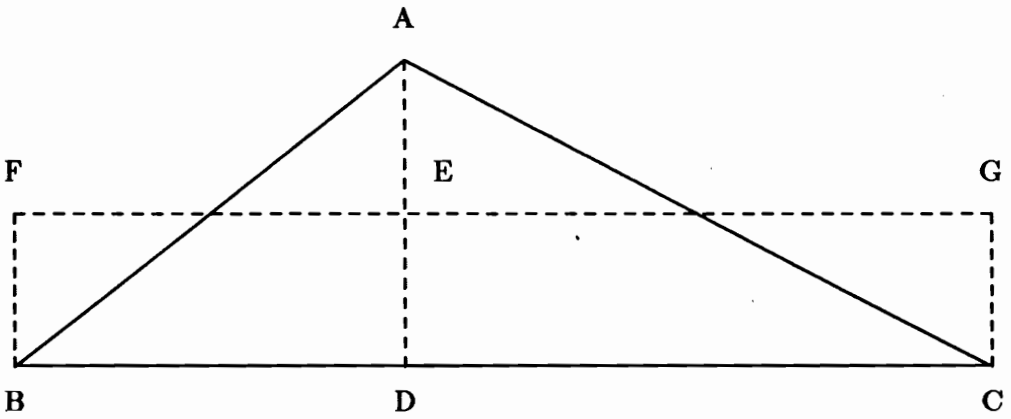


Figure 2: Given triangle ABC, drop perpendicular from A to D on BC, bisect AD at E, create rectangle FGCB with sides of lengths BC and ED = ½AD. The rectangle has the same area as the triangle.

We can make the following three observations about this solution:

The solution is not unique. We can choose any of three vertices and decide to bisect either the perpendicular or the opposite side. Indeed, by bisecting one side and doubling the other any number of times, we can construct rectangles with the correct area that are different from any of these solutions.

The solution is a set of instructions: do this, then that. That is, the solution is a program, as in *computer program*. The person carrying out the instructions is given no latitude in the solution method, and so the person plays the role of the computer.

The instructions of the solution are still above the level of ruler and compass, but perhaps they can be carried out using those tools. For example, some instructions are: "construct perpendicular," "bisect perpendicular," "construct rectangle." These instructions are *subprograms* which can be useful in solving many different problems.

The next level of construction detail might start:

1. Given a triangle, select any vertex and construct a perpendicular to the line that includes the opposite side.
   - 1.1 Select a compass setting such that an arc with its center at the vertex intersects the line twice.
   - 1.2 Bisect the line segment defined by the arc intersections.
   - 1.3 Construct a line segment from the vertex to the bisection point.

The next level of instructions might then continue:

- 1.2 Bisect the line segment defined by the arc intersections.
  - 1.2.1 Select a compass setting such that arcs with centers at the endpoints of the line segment intersect each other twice.

1.2.2 Connect the intersection points with a line segment.

1.2.3 The line segment just drawn in step 1.2.2 bisects the original line segment at the point of intersection.

Although the instruction to bisect a line segment has occurred twice (in Steps 1 and 1.2), it will only have to be described once. In a more complex construction, elementary instructions such as bisect, construct perpendicular, etc., will be used many times. In fact, a small set of such elementary instructions can be used in various combinations to define an unlimited set of constructions.

Imagine that a robot can read instructions and manipulate a ruler and compass to carry them out. The complete set of instructions for the robot must be very detailed, describing every motion of ruler, compass, and pencil. These detailed instructions are at the bottom of a hierarchy of higher-level instructions, whose top is the original problem statement. In this case, the task has been broken into three main parts (1, 2 and 3 above), then the first part broken into three parts (1.1, 1.2, and 1.3), and so on. (A Dewey decimal notation is convenient for numbering these parts of the hierarchy.)

The robot may not be capable of understanding higher-level instructions above the bottom of the hierarchy. They are of value only to the person who has to write out all the detailed instructions. The hierarchy allows this person to keep things straight. And if this person doesn't keep things straight, the robot will almost certainly go wrong. That is, if the high-level instructions say one thing but the detailed instructions say something else, the robot does the something else. So it is up to the person writing the program not only to have the right ideas but also to make sure that those ideas get translated step by step into the right details.

The best way to keep things straight is to build this hierarchy of high-level instructions from the top down, looking ahead all the time, and breaking each high-level instruction into a few next-level instructions. But before beginning to build this hierarchy, the program writer needs to understand the problem and find a solution strategy. Finding a solution strategy takes *creative thinking,* no doubt about it. But converting a solution strategy into a solution program, step by step, takes *creative discipline* as well.

CHAPTER 1

AN INTRODUCTION TO CF PASCAL

Chapter Preview

In Chapter 1 you will learn how to read simple programs written in CF Pascal, and how to predict the actions of a computer directed by these programs.

1.1 Pascal Programs

Preview

Simple Pascal programs have two parts: a program heading and a BEGIN statement. This section presents a simple Pascal program that has no input and has a one-time message as its output.

New ideas: Pascal program, Pascal machine, identifier, program part, program heading, BEGIN statement, WRITELN statement.

A *Pascal program* is a written set of statements for a computer to carry out. A Pascal program is written in the *Pascal programming language.* Another computer program (called a Pascal *compiler*) makes the conversion from statements in Pascal to a machine program that the computer can execute. Pascal programs have a precise form, which specifies a precise execution. We will use the term *Pascal machine* to mean any combination of computer hardware and software (usually including a Pascal compiler) that carries out the execution specified by a Pascal program. The exact procedures for invoking a Pascal machine depend on the specific computer and on the installation.

Program 1 shown below with input and output is a Pascal program of four lines. Program 1 will ignore any data given it; indeed, it will execute without any input data.

Program 1: PRINTHELLO

```
PROGRAM PRINTHELLO(INPUT,OUTPUT);
  BEGIN
    WRITELN('HELLO')
  END.
```

Execution

```
OUTPUT:HELLO
```

When executed, Program 1 will print the characters **HELLO** as its output.

The detailed form in which a program must appear is called its *syntax.* Program 1 begins with the word PROGRAM and ends with the word END and a period. This is part of the syntax of every Pascal program. The words PROGRAM, INPUT, OUTPUT, BEGIN, WRITELN, END are *standard words* in the Pascal language. In a program they must be spelled exactly as given. The punctuation marks

```
( , ) ; ' .
```

must also be properly placed. The standard Pascal word WRITELN directs the Pascal machine to write (print) a given line of data. The word INPUT is the name of the data to be used by the program as input (which is ignored by Program 1). The word OUTPUT is the name of the output data that will be printed as a result of the execution of this program. The words PROGRAM, BEGIN, and END notify the Pascal machine that a program exists to be executed. The final period signifies the end of the program. Notice that when standard Pascal words appear, they are set in a different type font, so they will not be confused with the words of a sentence.

The quoted letters 'HELLO' comprise a *character string.* Indeed, it is part of 'HELLO' that the program prints and not the HELLO that is part of the name PRINTHELLO.

The word PRINTHELLO, found right after the standard word PROGRAM, is the name of Program 1. The name of the program does not direct the Pascal machine in execution, but it should be descriptive to programmers, nevertheless. The word PRINTHELLO is called an *identifier* and is not a standard Pascal word, but because it appears in a program it is set in the same type font as Pascal words. The rules for forming Pascal identifiers are precise: An identifier must start with a letter and may contain additional letters or digits with no spaces or other characters intervening.

The line and indentation organization of a program do not effect execution. For example, the Pascal machine will treat the following program just as it does Program 1:

```
PROGRAM PRINTHELLO(INPUT,OUTPUT);
BEGIN WRITELN('HELLO') END.
```

The Pascal machine processes varying line and indentation organization with equal facility, but people do not. When used consistently, good line and indentation rules make Pascal programs much easier for people to read.

If the punctuation marks are deleted, the text

```
PROGRAM PRINTHELLO INPUT OUTPUT BEGIN WRITELN HELLO END
```

is no longer a Pascal program, just a list of words.

We can break a program into sections, called *program parts,* to improve comprehension. In Program 1, the program part

```
PROGRAM PRINTHELLO(INPUT,OUTPUT)
```

is called a *program heading.* The next program part,

```
BEGIN
  WRITELN('HELLO')
END
```

(it does not include the final period) is called a *BEGIN statement.* The BEGIN statement consists of the word BEGIN, the word END, and everything in between. The program part

```
WRITELN('HELLO')
```

is called a *WRITELN statement,* which instructs the Pascal machine to write the character string **HELLO** as a line of data to OUTPUT.

Program 2 will print **HELLO**, not **GOODBYE**, because the data 'HELLO' are provided in the WRITELN statement.

Program 2: PRINTGOODBYE

```
PROGRAM PRINTGOODBYE(INPUT,OUTPUT);
  BEGIN
    WRITELN('HELLO')
  END.
```

Execution

```
OUTPUT:HELLO
```

This example illustrates that the program name does not direct the Pascal machine. PRINTGOODBYE is a misleading name, but it is only a name and not data.

If Program 2 were written as:

```
PROGRAM PRINTGOODBYE(INPUT,OUTPUT); BEGIN WRITELN('HELLO') END.
```

it could be presented for easier reading in two steps:

Step one: Look for the standard words BEGIN and END. Start a new line at BEGIN; start another new line just after BEGIN; and start a third new line at END, to get

```
PROGRAM PRINTGOODBYE(INPUT,OUTPUT);
BEGIN
WRITELN('HELLO')
END.
```

Step two: Indent the BEGIN line two spaces; then indent the new line just after BEGIN two more spaces; and then indent the next line two spaces, to make END even with BEGIN, to get

```
PROGRAM PRINTGOODBYE(INPUT,OUTPUT);
  BEGIN
    WRITELN('HELLO')
  END.
```

which is exactly Program 2 given above. There is nothing special about indenting two spaces at a time, rather than three or more. But in complex programs, more than two spaces may be difficult to read.

In fact, if the words of Program 2 were given with the punctuation omitted, the resulting list of words

```
PROGRAM PRINTGOODBYE INPUT OUTPUT BEGIN WRITELN HELLO END
```

can be made into a program by inserting the correct punctuation marks. First, the standard words INPUT and OUTPUT must be separated by a comma, surrounded by parentheses, and the result followed by a semicolon:

```
(INPUT,OUTPUT);
```

Second, the characters following WRITELN must be surrounded by single quote marks and then by parentheses:

```
('HELLO')
```

Finally, END must be followed by a period:

```
END.
```

When these rules are followed, the list of words above becomes

```
PROGRAM PRINTGOODBYE(INPUT,OUTPUT); BEGIN WRITELN('HELLO') END.
```

which is exactly Program 2 given as a single string of characters above.

Unlike names (identifiers), which are made up of letters and digits, Pascal character strings can be made up of any characters, including blanks, as illustrated in Program 3.

Program 3: PRINTSENTENCE

```
PROGRAM PRINTSENTENCE(INPUT,OUTPUT);
  BEGIN
    WRITELN('TAKE ROUTE 17, THEN 452.')
  END.
```

Execution

```
OUTPUT:TAKE ROUTE 17, THEN 452.
```

Since the single quote mark is used in a special way to bracket data, some convention is needed to signify a quote mark or apostrophe in a character string, as in CAN'T. Two adjacent quote marks signify the presence a single quote mark or apostrophe, as shown in Program 4.

Program 4: PRINTWARNING

```
PROGRAM PRINTWARNING(INPUT,OUTPUT);
  BEGIN
    WRITELN('DON''T TAKE ROUTE 17.')
  END.
```

Execution

```
OUTPUT:DON'T TAKE ROUTE 17.
```

1.1.1 Exercises

1.1.1 For the following programs, give the name and output.

a)
```
PROGRAM HELLO(INPUT,OUTPUT);
  BEGIN
    WRITELN('GOODBYE')
  END.
```

b)
```
PROGRAM NEEDLE(INPUT,OUTPUT);
  BEGIN
    WRITELN('THE NEEDLE ISN''T IN THE HAYSTACK')
  END.
```

c)
```
PROGRAM SAYONARA(INPUT,OUTPUT);
  BEGIN
    WRITELN('''SAYONARA''')
  END.
```

1.1.2 Break the following programs into lines and indent the lines for good readability.

a)
```
PROGRAM PIN(INPUT,OUTPUT); BEGIN WRITELN
      ('I HEARD THE PIN DROP!') END.
```

b)
```
PROGRAM ANGELS(INPUT,OUTPUT); BEGIN
   WRITELN('''ANGELS CAN BE HEARD ON HIGH''') END.
```

c)
```
PROGRAM QUESTION(INPUT,
  OUTPUT); BEGIN
  WRITELN('ISN''T THIS EASY?')
    END.
```

1.1.3 Make the following texts into programs by inserting the needed punctuation, and break the programs into lines and indent the lines for good readability. What is the name and output of each program?

a)
```
PROGRAM WESTERN INPUT OUTPUT BEGIN WRITELN HOWDY END
```

b)
```
PROGRAM GERMAN INPUT OUTPUT BEGIN WRITELN GUTEN TAG END
```

c)
```
PROGRAM GOODBYE INPUT OUTPUT BEGIN WRITELN SOLONG END
```

1.2 Sequential Execution

Preview

Pascal programs are executed by the Pascal machine one statement at a time. Pascal programs can print character strings on the same line or on multiple lines.

New ideas: WRITE statement, execution table, continued printing on the same line, beginning a new line of output, printing shapes.

Pascal machines execute Pascal programs sequentially; that is, they carry out statements one at a time. Program 5 demonstrates sequential execution. The word HELLO is printed one or two characters at a time instead of all at once. In Program 5 a new standard word WRITE defines a WRITE statement that will write the data given but not close the line of data. The WRITELN

statement means to write the data given and to close the line of data for printing. The sequential steps of execution are *separated* by *semicolons*, and for readability, we start a new line for each statement. The BEGIN and END words are not separated from the statements between them by semicolons.

Program 5: PRINTHELLOTOO

```
PROGRAM PRINTHELLOTOO(INPUT,OUTPUT);
  BEGIN
    WRITE('HE');
    WRITE('L');
    WRITELN('LO')
  END.
```

semicolon means "and then"

Execution

```
OUTPUT:HELLO
```

It will be useful to visualize the state of OUTPUT at various points in the execution sequence. Just after the statement

```
WRITE('HE')
```

OUTPUT will contain the characters **HE**. Just after the statement

```
WRITE('L')
```

OUTPUT will contain **HEL**. That is, **L** is added to the line holding only **HE** up to that moment. Finally, just after the statement

```
WRITELN('LO')
```

OUTPUT will contain the whole word **HELLO**, and the line will be closed out for printing.

One way to describe sequential execution is by using an *execution table,* such as the following for Program 5:

Execution Table for Program 5

After Execution of	OUTPUT
PROGRAM PRINTHELLOTOO(INPUT,OUTPUT)	_
BEGIN	
WRITE('HE')	**HE**_
WRITE('L')	**HEL**_
WRITELN('LO')	**HELLO**/_
END.	**HELLO**

The Pascal machine executes one statement at a time, as shown in this execution table. After the first line (the program heading), OUTPUT is created to receive data during execution. OUTPUT will always be shown as a string of characters ending with an underline *cursor* that points to the next place a character may be written.

At the beginning of execution, there are no characters in OUTPUT, so the cursor begins the string. The second line (the word BEGIN) does not affect OUTPUT. But the third line writes (with WRITE) the two characters **HE** into OUTPUT, beginning at its cursor, and the cursor moves two places to the right. The fourth line writes **L** into OUTPUT at its cursor and moves the cursor again. Similarly, the fifth line writes **LO** (with WRITELN), and closes the line—shown here as a slash (/) character as a line marker—and moves the cursor. The last line (the word END and the period) terminates execution and precludes the possibility of writing any more data to OUTPUT. Thus its cursor and line marker disappear, leaving the character string **HELLO** as the output of the program.

The slash (/) character used to mark the end of lines (each time WRITELN is used) is not an actual character in OUTPUT. There is a character / that can be printed in Pascal, but we will not use it in examples, to avoid confusion with line ends. The tabular execution table separates the lines of a program, so the semicolons are omitted.

Summary of Execution Table for Program 5

	After execution of	OUTPUT
No characters in OUTPUT; cursor begins the string.	PROGRAM...	_
BEGIN does not affect OUTPUT.	BEGIN	
Writes **HE** into OUTPUT beginning at its cursor and moves the cursor two places to the right.	WRITE('HE')	**HE**_
Writes **L** into OUTPUT at its cursor and moves the cursor one more space to the right.	WRITE('L')	**HEL**_
Writes **LO** into OUTPUT, closes out the line, and moves the cursor.	WRITELN('LO')	**HELLO**/_
Terminates execution and precludes any further writing to OUTPUT.	END.	**HELLO**

Program 5 can be reconstructed from its list of words alone, by first forming proper WRITE or WRITELN statements; second, separating successive statements with semicolons; and finally, starting each statement on a new line, indented two spaces. That is, given the list of words

```
PROGRAM PRINTHELLOTOO INPUT OUTPUT BEGIN WRITE HE
WRITE L WRITELN LO END
```

the rules can give back

```
PROGRAM PRINTHELLOTOO(INPUT,OUTPUT);
  BEGIN
    WRITE('HE');
    WRITE('L');
    WRITELN('LO')
  END.
```

which is exactly Program 5. Different programs can be produced if the quoted strings within WRITE statements are different.

Strange as it may seem, a blank is a data character, so we can print a blank by itself, as illustrated in Program 6.

Program 6: PRINTGREETING

```
PROGRAM PRINTGREETING(INPUT,OUTPUT);
  BEGIN
    WRITE('HELLO');
    WRITE(' ');
    WRITELN('THERE')
  END.
```

Execution

```
OUTPUT:HELLO THERE
```

The execution of Program 6 can be described by a table as before.

Execution Table for Program 6

After Execution of	OUTPUT
`PROGRAM PRINTGREETING(INPUT,OUTPUT)`	_
`  BEGIN`	
`    WRITE('HELLO')`	**HELLO**_
`    WRITE(' ')`	**HELLO** _
`    WRITELN('THERE')`	**HELLO THERE**/_
`  END.`	**HELLO THERE**

In Program 7, there is more than one line of output.

Program 7: VERTICALGREETING

```
PROGRAM VERTICALGREETING(INPUT,OUTPUT);
  BEGIN
    WRITELN('HELLO');
    WRITELN('THERE')
  END.
```

Execution

```
OUTPUT:HELLO
       THERE
```

Execution Table for Program 7

After Execution of	OUTPUT
`PROGRAM VERTICALGREETING(INPUT,OUTPUT)`	_
`  BEGIN`	
`    WRITELN('HELLO')`	**HELLO**/_
`    WRITELN('THERE')`	**HELLO**/**THERE**/_
`  END.`	**HELLO**
	THERE

Within this execution table the output has two line markers (shown as slashes), because each WRITELN statement moves the cursor to the beginning of a new line. After the first WRITELN statement it is not possible to add any characters on the line with **HELLO**.

WRITELN can be used without any data to close the old line and start a new one. That is, WRITELN allows data to be organized by lines. Indenting can be accomplished by writing blanks. To illustrate this, Program 8 prints an indented greeting.

Program 8: INDENTGREETING

```
PROGRAM INDENTGREETING(INPUT,OUTPUT);
  BEGIN
    WRITE('HELLO');
    WRITELN;
    WRITE('      ');
    WRITE('THERE');
    WRITELN
  END.
```

Execution

```
OUTPUT:HELLO
             THERE
```

WRITE and WRITELN can be combined to produce printed patterns. The name of Program 9, Cross, is the first to use lowercase letters. Pascal admits both cases, but not all Pascal machines have the ability to use them.

Program 9: Cross

```
PROGRAM Cross(INPUT,OUTPUT);
  BEGIN
    WRITELN('  C  ');
    WRITELN('  R  ');
    WRITELN('CROSS');
    WRITELN('  S  ');
    WRITELN('  S  ')
  END.
```

Execution

```
OUTPUT:  C
         R
       CROSS
         S
         S
```

In the rest of this book, standard words will be given in uppercase letters, while invented names will start with an uppercase letter followed by any letters or digits. The characters printed by Program 9 are all in uppercase, but if the Pascal machine allows it, lowercase could be printed as well.

One practical and satisfying use of computers is to print large block letters.

Program 10: BlockXYZ

```
PROGRAM BlockXYZ(INPUT,OUTPUT);
  BEGIN
    WRITELN('X   X Y   Y ZZZZZ');
    WRITELN(' X X   Y Y     Z ');
    WRITELN('  X     Y     Z  ');
    WRITELN(' X X    Y    Z   ');
    WRITELN('X   X   Y   ZZZZZ')
  END.
```

Execution

```
OUTPUT:X   X Y   Y ZZZZZ
        X X   Y Y     Z
         X     Y     Z
        X X    Y    Z
       X   X   Y   ZZZZZ
```

1.2.1 Exercises

1.2.1 Determine the execution tables for:

a) Program 8, INDENTGREETING

b) Program 9, Cross

1.2.2 Break the programs listed into lines, indent the lines for good readability, and determine the output.

a)
```
PROGRAM Poem(INPUT,OUTPUT); BEGIN WRITE
('ROSES ARE '); WRITELN('RED,');
WRITE(' VIOL'); WRITE('ETS ARE'); WRITELN
(' BLUE '); WRITE('OTHERS CAN PR'); WRITELN
('OGRAM,'); WRITE(' SO CAN'); WRITELN(' YOU') END.
```

b)
```
PROGRAM Pattern1(INPUT,OUTPUT); BEGIN WRITE('  *');
WRITELN; WRITELN(' * *'); WRITE('*   *'); WRITELN;
WRITE(' * *'); WRITELN; WRITELN(' *') END.
```

c)
```
PROGRAM Pattern2(INPUT,OUTPUT); BEGIN WRITE
(' * '); WRITELN; WRITELN('* *'); WRITELN
(' * '); WRITE('* '); WRITELN('*');
WRITELN('* *'); WRITELN(' * ') END.
```

1.2.3 Make the following texts into readable programs by inserting punctuation marks and indenting properly. Determine the output of each program.

a)
```
PROGRAM ShortGreeting INPUT OUTPUT
BEGIN WRITELN Hi! END
```

b)
```
PROGRAM Palindrome INPUT OUTPUT
BEGIN WRITE MADAM WRITELN IM ADAM END
```

c)
```
PROGRAM Mozart INPUT OUTPUT BEGIN WRITELN
TWINKLE TWINKLE LITTLE STAR WRITELN
HOW I WONDER WHAT YOU ARE END
```

1.2.4 In programs 9 and 10, all the WRITELN statements have the same number of characters between quotes (so that the appearance of the printed information lines up in the program). Which blanks included between the quotes are unnecessary, and why?

1.3 Character Data

Preview

The simple Pascal programs in Sections 1.1 and 1.2 used WRITE and WRITELN statements to produce output. In many programs it is necessary to read data values into program variables for use in the program. Once a value has been read into a variable, it can be copied into other variables or printed out.

New ideas: READ statement, VAR, CHAR, character-valued variables, identifiers, declaration part, INPUT, normal execution, aborted execution, assignment statement, assignment operator, character literal, comments, annotated output, echoed input.

A ***character-valued variable*** can be defined by a new program part, called a ***declaration*** part, that is headed by the standard word VAR (for VARiables). This declaration part contains an identifier followed by a colon, and the standard word CHAR (for CHARacter). A semicolon separates the declaration part from the BEGIN statement that follows it. For example,

```
VAR
  Ch: CHAR
```

is a declaration part in which the identifier is Ch. This name Ch is often called a ***variable,*** and it can be thought of as naming a value. Because CHAR follows the colon, the value will be a character. We say that the variable Ch is declared to be of ***type*** CHAR in this declaration part, which is used in Program 11 below.

In Program 11 the second line enclosed in braces { } is called a ***comment.*** It has no effect on the Pascal machine, but can be used to place information about a program with the text so that it will not be forgotten. Here the program meaning is not obvious at a glance, so it will not hurt to remind ourselves of it by looking at the comment.

Program 11: CopyChar

```
PROGRAM CopyChar(INPUT,OUTPUT);
  {Writes the first character in INPUT to OUTPUT}
  VAR
    Ch: CHAR;
  BEGIN
    READ(Ch);
    WRITELN(Ch)
  END.
```

Execution

```
INPUT :ABC
OUTPUT:A
```

The CHAR variable Ch can take on values that are single characters only—not values that are several characters at once. In Program 11, the standard word READ defines a *READ statement* that refers to variable Ch. The READ statement sets Ch to the value of the next character found in INPUT. In turn, the WRITELN statement will place the character value to which Ch is set, in OUTPUT. Notice that Ch is not surrounded by quotes in the WRITELN statement. This ability to transfer variable values to OUTPUT is another capability of WRITELN.

The execution of Program 11 is described by the table below, in which there is a column for the value of variables Ch and INPUT as well as for OUTPUT.

Execution Table for Program 11

After Execution of	INPUT	OUTPUT	Ch
	ABC		
PROGRAM CopyChar(INPUT,OUTPUT)	**ABC**	_	
VAR			
Ch: CHAR	**ABC**	_	?
BEGIN			
READ(Ch)	**ABC**	_	**A**
WRITELN(Ch)	**ABC**	**A**/_	**A**
END.	**ABC**	**A**	

In addition to the new columns, this execution table has a new first row showing the initial data in INPUT. INPUT holds the character string **ABC** before program execution begins. The underline cursor for INPUT is created by the first line of the program, and it marks the first character of the input string **ABC** as ready to read. The variable Ch does not exist until after the execution of the third line of the program. When it comes into existence through the declaration, Ch can start off being any value (for reasons we explain a little later). The question mark under Ch does not mean the character **?** but indicates a state of ignorance: Ch can be literally any character value at this point. READ(Ch) affects both INPUT and Ch. It gives the value at the cursor of INPUT (namely **A**) to Ch and simultaneously moves the cursor to the next character of INPUT. The WRITELN(Ch) command writes the current value of Ch (in this case, **A**) to OUTPUT and moves the OUTPUT cursor. After the final line, the cursors on INPUT and OUTPUT disappear, as does the variable Ch.

The only record of execution that remains is the two files. Of course, INPUT contains no new information, but OUTPUT contains the results of the execution. There is no trace of the variable Ch, which exists only within the program.

When a program takes input, it may be useful to examine several executions with different input strings, which will be shown in a group following the program. For Program 11, for example:

Program 11: CopyChar

```
PROGRAM CopyChar(INPUT,OUTPUT);
  {Writes the first character in INPUT to OUTPUT}
  VAR
    Ch: CHAR;
  BEGIN
    READ(Ch);
    WRITELN(Ch)
  END.
```

Executions

```
INPUT :ABC
OUTPUT:A

INPUT :A
OUTPUT:A

INPUT :XAABB
OUTPUT:X
```

You should read the executions section as: if the input is **ABC**, then the output will be **A**; if the input is **A**, the output will be **A**, etc. These different cases do not occur together, but each represents one execution of the program.

In Program 12, the sequence of READ/WRITELN in Program 11 is reversed.

Program 12: Garbage

```
PROGRAM Garbage(INPUT,OUTPUT);
  {It is a mistake to write before reading.}
  VAR
    Ch: CHAR;
  BEGIN
    WRITELN(Ch);
    READ(Ch)
  END.
```

Executions

```
INPUT :ABC
OUTPUT:?

INPUT :A
OUTPUT:?

INPUT :XYZ
OUTPUT:?
```

The question mark output in Program 12 means that there is a character in OUTPUT but this character is entirely unpredictable, having nothing to do with the input. The WRITELN statement will be executed by the Pascal machine before the READ statement. The value of Ch has not been set, so anything may appear in the output. In a typical Pascal machine this value will be one left in storage before execution began. Data with no sensible origin is usually called "garbage." Following the WRITELN statement, the READ statement will place the first value from INPUT in Ch, but too late to be used, as the execution table below shows:

Execution Table for Program 12

After Execution of	INPUT	OUTPUT	Ch
	A		
PROGRAM Garbage(INPUT,OUTPUT)	**A**	_	
VAR			
Ch: CHAR	**A**	_	?
BEGIN			
WRITELN(Ch)	**A**	?/_	?
READ(Ch)	**A**_	?/_	**A**
END.	**A**	?	

There is a fundamental lesson in this simple example. The Pascal machine executes instructions to the letter; the programmer's intention must be represented precisely with instructions. If a programmer is not careful, then some data can be lost and other data inadvertently reused.

Program 13 provides more practice in seeing exactly what the Pascal machine will do. It might appear that CopyTwice would copy two characters from INPUT to OUTPUT, but it is not so.

Program 13: CopyTwice

```
PROGRAM CopyTwice(INPUT,OUTPUT);
  {A mistaken attempt to copy two characters.}
  VAR
    Ch: CHAR;
  BEGIN
    READ(Ch);
    READ(Ch);
    WRITE(Ch);
    WRITELN(Ch)
  END.
```

Executions

```
INPUT :ABC
OUTPUT:BB
INPUT :XXY
OUTPUT:XX
```

) bad test

Program 13's instructions are literally to transfer the first character from INPUT to Ch, and then to overlay that value by transferring the second character from INPUT, also to Ch. Thus when two characters are subsequently written, they are one and the same (the second character from INPUT). Notice what a bad test the second execution is: if the programmer is expecting a two-character copy, that execution might confirm this wrong idea.

Program 14 shows how to correctly copy two characters from INPUT to OUTPUT.

Program 14: Copy2

```
PROGRAM Copy2(INPUT,OUTPUT);
  {Copies the first two characters from INPUT to OUTPUT}
  VAR
    Letter: CHAR;
  BEGIN
    READ(Letter);
    WRITE(Letter);
    READ(Letter);
    WRITELN(Letter)
  END.
```

Executions

```
INPUT :ZZ
OUTPUT:ZZ
INPUT :AZURE
OUTPUT:AZ
INPUT :1234
OUTPUT:12
INPUT :3.75
OUTPUT:3.
```

The variable `Letter` holds different values at different times, sometimes letters and sometimes other characters, as befits its declaration as `CHAR`. As with program names, it is useful to invent descriptive variable names, but don't be deceived by them.

Programs can have one, two, ten, or more variables. The two variables in Program 15, `Ch1` and `Ch2`, appear in the declaration part and the READ and WRITE statements.

Program 15: CopyReversed

```
PROGRAM CopyReversed(INPUT,OUTPUT);
  {The first two characters from INPUT are
   copied to OUTPUT in reverse order}
  VAR
    Ch1, Ch2: CHAR;
  BEGIN
    READ(Ch1, Ch2);
    WRITELN(Ch2, Ch1)
  END.
```

Executions

```
INPUT :ZA
OUTPUT:AZ
INPUT :AZURE
OUTPUT:ZA
```

When READ and WRITE statements are given more than one data item, they behave like sequences of READ and WRITE statements with single data items. For example

```
READ(Ch1, Ch2)
```

has the same effect as the sequence

```
READ(Ch1); READ(Ch2)
```

and similarly,

```
WRITE('Value is: ', Ch1)
```

has the effect of

```
WRITE('Value is: '); WRITE(Ch1)
```

A WRITELN statement with data item(s) behaves like a WRITE statement followed by a bare WRITELN statement.

```
WRITELN(Ch1, 'THEN', Ch2)
```

has the same effect as

```
WRITE(Ch1, 'THEN', Ch2); WRITELN
```

or as above:

```
WRITE(Ch1); WRITE('THEN'); WRITE(Ch2); WRITELN
```

There are evidently many ways to group items in READ and WRITE statements; WRITELN statements must be used to obtain the desired line structure. For example,

```
WRITELN(Ch1, Ch2, Ch3); WRITELN(Ch4, Ch5)
```

has the same effect as

```
WRITE(Ch1, Ch2); WRITELN(Ch3); WRITE(Ch4); WRITELN(Ch5)
```

and both produce two lines, the first of three values, the second of two.

It is easy to see how to extend Program 15 to copy three characters in reverse order, four variables in reverse order, etc., by declaring enough variables and then putting them in reverse order in the READ and WRITE lists.

1.3.1 Normal Execution and Aborted Execution

An execution is called *normal* if and only if it reaches the final period of the program. A program including READ statements has the potential to *abort* if it attempts to obtain characters from INPUT when they are not present. If a READ statement is attempted when no characters are available, the program is terminated and a message is printed saying what happened. For example, suppose a program contains the READ statement

```
READ(Ch1, Ch2, Ch3)
```

which is intended to obtain the first three characters on INPUT. If INPUT contains too few characters, the program will abort.

There is a complication when input is divided into lines, which will be described in detail in Section 4.3. At the end of each line there appears to be one additional character, whose value is blank. Thus the three-character READ statement above, given as INPUT one line containing the two characters **XY**, will set Ch1 to **X**, Ch2 to **Y**, and Ch3 to blank. Given the single-character INPUT **Z**, the program containing this READ statement will abort. For some Pascal machines, output written before an abort is printed ahead of the abort message; for some Pascal machines this partial output is discarded. When partial output is lost, the programmer may be misled about what has taken place in the program.

Aborted execution does not alter or damage the program, which may sensibly be executed again with INPUT having enough data. But whenever a program contains READ statements the possibility of too little data exists, and the program may abort.

1.3.2 The Assignment Statement

Up to this point, variables have been used to store data that are read from INPUT. The only way to use data has been to write the variable values to OUTPUT. Values may also be copied from one variable to another. The copying process is called ***data assignment*** and is written in an ***assignment statement,*** which has the following forms:

Name1 := *Name2*
Name1 := '*x*'

Think of an arrow '⇐'

where *Name1* and *Name2* are CHAR variables declared in the VAR section, and where *x* is any character. The special double character := is called the *assignment operator*. In the first form of the assignment statement, the value of the variable on the right side (*Name2*), is assigned as the new value of the variable on the left side (*Name1*). The value of *Name2* is not affected by the assignment. In the second form of the assignment statement, *Name1* is assigned the *character literal x*. Assignment is not defined for files. The text

```
OUTPUT := INPUT
```

→ i.e. not allowed, or will work incorrectly

is not an assignment statement.

Assignment statements offer another way to reverse two characters.

Program 16: CopyReversedToo

```
PROGRAM CopyReversedToo(INPUT,OUTPUT);
  {The first two characters from INPUT are
   transferred to OUTPUT in reverse order}
  VAR
    In1, In2, Out1, Out2: CHAR;
  BEGIN
    READ(In1, In2);
    Out1 := In2;
    Out2 := In1;
    WRITELN(Out1, Out2)
  END.
```

Execution

```
INPUT :AZURE
OUTPUT:ZA
```

The assignment statement

```
Out1 := In2
```

is read "Out1 becomes in value the value of In2," or for short, "Out1 becomes In2."

Execution Table for Program 16

After Execution of	INPUT	OUTPUT	In1	In2	Out1	Out2
	AZURE					
PROGRAM CopyRev...	**AZURE**	_				
VAR						
In1, In2, ...	**AZURE**	_	?	?	?	?
BEGIN						
READ(In1, In2)	**AZURE**	_	**A**	**Z**	?	?
Out1:=In2	**AZURE**	_	**A**	**Z**	**Z**	?
Out2:=In1	**AZURE**	_	**A**	**Z**	**Z**	**A**
WRITELN(Out1, Out2)	**AZURE**	**ZA/_**	**A**	**Z**	**Z**	**A**
END.	**AZURE**	**ZA**				

Program 17 introduces two ideas that markedly improve the readability of program results. ***Annotated output*** contains text to help explain the output. ***Echoed input*** is a copy of the input, transferred to the output. When input is echoed back, the need to keep separate records correlating inputs with outputs is eliminated. If any trouble arises in program execution, it is very convenient to have input and output together for inspection and analysis.

Program 17: CopyAndShuffle

```
PROGRAM CopyAndShuffle(INPUT,OUTPUT);
  {The first five characters U, V, W, X, Y of INPUT
   are copied to OUTPUT, in the order U, X, V, Y, W.
   For four input characters, Y appears to have been blank.
   The program aborts if INPUT contains three or fewer
   characters.}
  VAR
    In1, In2, In3, In4, In5: CHAR;
  BEGIN
    READ(In1, In2, In3, In4, In5);
    WRITELN('INPUT DATA');
    WRITE('  ');
    WRITELN(In1, In2, In3, In4, In5);
    WRITELN('SHUFFLED DATA');
    WRITE('  ');
    WRITELN(In1, In4, In2, In5, In3)
  END.
```

Executions

```
INPUT :12345678
OUTPUT:INPUT DATA
         12345
       SHUFFLED DATA
         14253

INPUT :BARE
OUTPUT:BEA R
```

Notice that the comment describing Program 17 is very specific about the kind of input that is acceptable.

1.3.3 Exercises

1.3.1 Explain the difference between

```
WRITELN(Ch)
```

and

```
WRITELN('Ch')
```

If you saw one of these in a program, and suspected it was meant to be the other, what part of the program might provide a clue to the programmer's intent?

1.3.2 Determine the execution tables for the following programs with the input given.

a) Program 14, Copy2. INPUT:**ABC**

b) Program 15, CopyReversed. INPUT:**ABC**

1.3.3 Break the following program into properly indented lines, find a better program name, insert OUTPUT annotations, and describe what it does.

```
PROGRAM Harry(INPUT,OUTPUT); VAR First,
Second, Third: CHAR; BEGIN READ(First,
Second, Third); Third := Second; WRITELN
(Third) END.
```

1.3.4 Punctuate the given texts properly into programs, break into lines, indent, find a better name, insert OUTPUT annotation, and describe what the programs do.

a)
```
PROGRAM Print2 INPUT OUTPUT VAR First
Second Third CHAR BEGIN READ First Second
Third Second := First WRITELN Second END
```

b)
```
PROGRAM PrFirst INPUT OUTPUT VAR First
Second Third CHAR BEGIN READ First Second
Third First := Second WRITELN First END
```

c)
```
PROGRAM PrLast INPUT OUTPUT VAR First
Second Third CHAR BEGIN READ First Second
Third WRITELN Second END
```

1.3.5 Find out exactly what happens on your Pascal machine when you read past the available data. Things to try: reading the extra blank at the end of data; `INPUT` containing *no* data at all; `WRITE` with or without `WRITELN` before abort.

1.3.6 Why is

```
Ch2 := 'Ch1'
```

'Ch1' more than one char.

not a valid assignment statement? Under what conditions could

```
Ch2 := Ch1
```

Ch2 not declared

not be a valid assignment statement?

1.3.7 Explain why it is impossible to write a program with the same effect as Program 15: `CopyReversed` that uses only *one* variable. Is it possible to accomplish the task of Program 16: `CopyReversedToo` using fewer variables? Explain.

1.3.8 Why is the annotation **`INPUT DATA`** in Program 17 misleading? Improve the annotation.

1.3.9 Find an input of English text such that Program 17 will appear *not* to shuffle it—that is, the output and input will appear to be the same. Why is data of this kind useful or not useful in testing a program?

1.4 Conditional Execution

Preview

Choosing between two alternatives is vital in programming a computer. A character variable can be compared with a character literal or another character variable. Then the results of this comparison can be used to direct the Pascal machine to execute one of two Pascal statements.

New ideas: precedes, succeeds, conditions, IF statement, BEGIN statement, component statement, null statement, maximum character, minimum character.

Many familiar procedures contain conditional instructions. Tax forms specify that "if you are self-employed, you must fill out Schedule C" or that "if your income is under $12,000, use the 1040S short form, but if your income is $12,000 or above, use the 1040 regular form." Vending machines are built so that if the input is 40 cents for a 35-cent item, a nickel is returned and if the input is 50 cents, a dime and a nickel are returned. These conditional execution situations are frequent parts of most programs and must be specified in a programming language. Pascal programs provide for *conditional execution*. That is, the decision to perform one statement or another during execution, depending on the outcome of a test.

1.4.1 Conditions

The Pascal machine has an ordering of the alphabet among either upper- or lowercase letters (when both are available) and among digits. This ordering is given by the *precedes symbol* (<), such that:

```
A < B < ... < Z
a < b < ... < z
0 < 1 < ... < 9
```

However, the ordering between characters in different lines above and the other characters depends on the Pascal machine. You should find out the exact character set and its ordering for the particular Pascal machine you are using.

Given two character variables Ch1 and Ch2, we can form a ***condition*** by asking: does the value of Ch1 precede the value of Ch2? Or, using the precedes symbol: Ch1 < Ch2 ? The answer depends, of course, on what the present values of Ch1 and Ch2 are. We give some examples in the following table:

Ch1	Ch2	Ch1 < Ch2
A	B	yes
B	A	no
A	A	no
1	9	yes
3	2	no

Conditions can also be used to test the equality of two characters using the ***equality symbol*** (=), as in the following table:

Ch1	Ch2	Ch1 = Ch2
A	B	no
A	A	yes
1	2	no

The ***succeeds symbol*** (>) means the opposite of precedes; that is, if Ch1 > Ch2 means Ch2 < Ch1. Examples:

Ch1	Ch2	Ch1 > Ch2
C	D	no
Z	A	yes
9	0	yes

Precedes or succeeds can be combined with equality, as in the question: Is Ch1 <= Ch2 ? Read this as "Does Ch1 precede or equal Ch2 ?" Some examples are presented in the following two tables:

Ch1	Ch2	Ch1 <= Ch2
A	B	yes
B	A	no
A	A	yes
1	1	yes

Ch1	Ch2	Ch1 >= Ch2
A	B	no
B	A	yes
A	A	yes
1	1	yes

The double symbol <> means ***not equal,*** as in the following table:

Ch1	Ch2	Ch1 <> Ch2
A	A	no
A	B	yes
B	A	yes

The double symbols <=, >=, and <> have no spaces in them. These double symbols cannot be reversed; =<, =>, >< are meaningless in Pascal.

1.4.2 The IF Statement

Conditional execution is defined by the *IF statement*—a statement with the standard Pascal words IF, THEN, and ELSE in the following form:

```
IF condition
THEN
    THEN statement
ELSE
    ELSE statement
```

where *THEN statement* and *ELSE statement* are other Pascal statements (such as assignment statements). If the *condition* is satisfied, the *THEN statement* is executed; otherwise (if the condition is not satisfied), the *ELSE statement* is executed.

As a rule, an IF statement should be written in the style above, with the IF, THEN, ELSE lined up, and the text for the statements indented by two spaces. Such a standard style makes reading easier and more accurate. Program 18 illustrates the use of an IF statement.

Program 18: MaxChar

```
PROGRAM MaxChar(INPUT,OUTPUT);
  {The larger of the first two characters
   in INPUT  (in the sense of  > )
   is transferred to OUTPUT.  The program aborts
   if INPUT is empty, and the results are
   uncertain if INPUT has but one character}
  VAR
    Ch1, Ch2, Max: CHAR;
  BEGIN
    READ(Ch1, Ch2);
    IF Ch1 > Ch2
    THEN
      Max := Ch1
    ELSE
      Max := Ch2;
    WRITELN('MAX OF ', Ch1, ', ', Ch2,
            ' IS ', Max, '.')
  END.
```

Executions

```
INPUT :AZURE
OUTPUT:MAX OF A, Z IS Z.

INPUT :1234
OUTPUT:MAX OF 1, 2 IS 2.

INPUT :77
OUTPUT:MAX OF 7, 7 IS 7.
```

The form of this output, in which input is echoed and annotations are included, is a very common one. A semicolon separates the IF statement from the READ statement preceding it and the WRITELN statement following it, but there are no semicolons within the IF statement. In Program

18, the first execution is described by the table below, which suppresses some of the information of a full execution table in order to focus on the effect of the IF statement.

Partial Execution Table for Program 18

After Execution of	Condition	INPUT	Ch1	Ch2	Max
BEGIN		**AZURE**	?	?	?
READ(Ch1, Ch2)		**AZURE**	**A**	**Z**	?
IF Ch1 > Ch2	no				
ELSE					
Max := Ch2		**AZURE**	**A**	**Z**	**Z**
WRITELN		**AZURE**	**A**	**Z**	**Z**
END.		**AZURE**			

The absence of THEN and the presence of ELSE in this partial execution table means that the ELSE statement was executed.

The ELSE word and statement can be omitted from an IF statement, in this form:

IF *condition*
THEN
    THEN statement

In this case, if ***condition*** **is not satisfied, then nothing is done in the IF statement and the statement following it is executed.**

Program 19 illustrates an alternative program with identical behavior to that of Program 18, but using an IF statement without an ELSE. After the IF statement, Ch1 will be the maximum of its original value and the value of Ch2. If it happened that the value of Ch1 was already the maximum, nothing needs to be done; on the other hand, if the value in Ch2 is the maximum, Ch1 is given this larger value in the THEN statement. Ch1 and Ch2 are echoed before the IF statement, so that the original Ch1 value can be discarded, if need be, in the IF statement. Program 19 does not need the variable Max.

Program 19: MaxToo

```
PROGRAM MaxToo(INPUT,OUTPUT);
  {The larger of the first two characters
   in INPUT  (in the sense of  > )
   is transferred to OUTPUT.  The program aborts
   if INPUT is empty, and the results are
   uncertain if INPUT has but one character}
  VAR
    Ch1, Ch2: CHAR;
  BEGIN
    READ(Ch1, Ch2);
    WRITE('MAX OF ', Ch1, ', ', Ch2, ' IS ');
    IF Ch1 < Ch2
    THEN
      Ch1 := Ch2;
    WRITELN(Ch1, '.')
  END.
```

Execution

```
INPUT :AZURE
OUTPUT:MAX OF A, Z IS Z.
```

1.4.3 The BEGIN Statement

Conditional execution applies not only to simple assignment statements but also IF statements. The THEN statement or ELSE statement may be expanded to a *BEGIN statement,* a sequence of

statements enclosed by a BEGIN-END pair and separated by semicolons. The individual component statements of a BEGIN statement may be of any type, even other BEGIN statements, which may themselves contain BEGIN statements, and so on. In a conditional execution, a BEGIN statement is executed either in its entirety or not at all, depending on the condition.

Program 20: MaxMin

```
PROGRAM MaxMin(INPUT,OUTPUT);
  {The larger of the first two characters
   in INPUT  (in the sense of  > )
   is transferred to OUTPUT, followed by
   the smaller character.  The program aborts
   if INPUT is empty, and the results are
   uncertain if INPUT has but one character}
  VAR
    Ch1, Ch2: CHAR;
  BEGIN
    READ(Ch1, Ch2);
    WRITELN('MAXMIN OF ', Ch1, ', ', Ch2);
    IF Ch1 > Ch2
    THEN
      BEGIN
        WRITELN(Ch1);
        WRITELN(Ch2)
      END
    ELSE
      BEGIN
        WRITELN(Ch2);
        WRITELN(Ch1)
      END
  END.
```

Executions

```
INPUT :AZURE
OUTPUT:MAXMIN OF A, Z
       Z
       A

INPUT :1234
OUTPUT:MAXMIN OF 1, 2
       2
       1

INPUT :wcFields
OUTPUT:MAXMIN OF w, c
       w
       c
```

As a rule, a BEGIN statement should be written in the style above, with the BEGIN, END lined up and the component statements indented by two spaces. Such a standard style makes reading easier and more accurate.

The group of statements in a BEGIN statement may be empty. The statement that isn't there occurs frequently enough to have a name: the ***null statement***.

```
BEGIN
END
```

is obviously null. But examine the following carefully:

```
BEGIN
  Ch1 := Ch2;
END
```

Because of the semicolon separator, there are two component statements: an assignment before the semicolon, and a null statment after the semicolon.

A null statement proves useful in certain circumstances. For example, the IF statement

```
IF Ch1 < Ch2
THEN
ELSE
  Ch1 := Ch2
```

has a null statement for its THEN statement. This means "Do nothing if `Ch1 < Ch2`; otherwise, ..."

IF statements can be used as THEN statements or ELSE statements. In a pattern of IF statements inside IF statements, the Pascal machine associates each ELSE with the most recent THEN. In Program 21, two successive comparisons are needed to determine the maximum of three variables.

Program 21: Max3

```
PROGRAM Max3(INPUT,OUTPUT);
  {The largest of the first three characters
   from INPUT is transferred to OUTPUT.  For
   predictable results INPUT must contain at
   least three characters.}
  VAR
    Ch1, Ch2, Ch3: CHAR;
  BEGIN
    READ(Ch1, Ch2, Ch3);
    WRITE('MAX OF ', Ch1, ', ', Ch2,
          ', ', Ch3, ' IS ');
    IF Ch1 > Ch2
    THEN
      IF Ch1 > Ch3
      THEN
        WRITE(Ch1)
      ELSE
        WRITE(Ch3)
    ELSE
      IF Ch2 > Ch3
      THEN
        WRITE(Ch2)
      ELSE
        WRITE(Ch3);
    WRITELN('.')
  END.
```

Standard Pascal?

Executions

```
INPUT :AZURE
OUTPUT:MAX OF A, Z, U IS Z.
INPUT :1234
OUTPUT:MAX OF 1, 2, 3 IS 3.
```

1.4.4 Exercises

1.4.1 Determine the execution table for the following programs using the given input.

a) Program 20, MaxMin; INPUT:**ABC**

b) Program 21, Max3; INPUT:**ABC**

1.4.2 Break the following programs into lines, indent properly, and say what the program does.

a)
```
PROGRAM Dup(INPUT,OUTPUT); VAR Ch1,
Ch2, Ch3: CHAR; BEGIN READ(Ch1, Ch2,
Ch3); IF Ch1 = Ch2 THEN WRITE(Ch1,
Ch2, ' '); IF Ch1 = Ch3 THEN WRITE
(Ch1, Ch3, ' '); IF Ch2 = Ch3 THEN
WRITE(CH2, Ch3); WRITELN END.
```

b)
```
PROGRAM Order(INPUT,OUTPUT); VAR Ch1,
Ch2, Ch3, Ch4: CHAR; BEGIN READ(Ch1,
Ch2, Ch3, Ch4); IF Ch1 > Ch2 THEN IF
Ch3 > Ch4 THEN WRITELN('YES') ELSE
WRITELN('NO') ELSE IF Ch3 < Ch4 THEN
WRITELN('YES') ELSE WRITELN('NO') END.
```

1.4.3 Punctuate the given texts into programs, break into lines, indent, and say what the program does.

a)
```
PROGRAM Which INPUT OUTPUT VAR Ch1 Ch2
Ch3 CHAR BEGIN READ Ch1 Ch2 Ch3 WRITE
Answer is IF Ch1 < Ch2 THEN IF Ch1 < Ch3
THEN WRITE Ch1 ELSE WRITE Ch3 ELSE IF
Ch2 < Ch3 THEN WRITE Ch2 ELSE WRITE Ch3
WRITELN ' ' END
```

b)
```
PROGRAM What INPUT OUTPUT VAR Ch CHAR
BEGIN READ Ch IF '0' <= Ch THEN IF Ch
<= '9' THEN WRITELN YES ELSE WRITELN NO
ELSE WRITELN NO END
```

c)
```
PROGRAM MandM INPUT OUTPUT VAR Ch1 Ch2
CHAR BEGIN READ Ch1 Ch2 IF Ch1 < Ch2 THEN
WRITELN Ch1 Ch2 ELSE WRITELN Ch2 Ch1 END
```

1.4.4 Consider the following program fragment:

```
IF 'A' <= 'B'
THEN
  WRITELN('Yes.')
ELSE
  WRITELN('No.')
```

Write another, simpler fragment that cannot be distinguished from this one in execution.

1.4.5 Suppose that there is inadequate documentation on your Pascal machine, in particular that you cannot learn the ordering of some characters, say - and #. Write a program that will determine the order of these two. Explain how one of programs 18 to 21 could have been used to do this job.

1.4.6 In program 21, suppose every ">" symbol were changed to ">=" to form program 21′. Discuss the difference between the operation of programs 21 and 21′.

1.4.7 In program 21, suppose every ">" symbol were changed to "<" to form program 21′. Discuss the difference in operation between programs 21 and 21′. What additional change should be made in program 21′?

1.4.8 Suppose that *A* and *B* are conditions, and *S* and *T* are statements. Compare the fragments of program:

```
IF A
THEN
  S;
IF B
THEN
  T
```

and

```
IF A
THEN
  S
ELSE
  IF B
  THEN
    T
```

Describe cases in which they behave differently.

1.5 Iterative Execution

Preview

It is essential for a computer program to execute statements repeatedly. The repetition is controlled by a condition that must be satisfied if the statements are to be repeated.

New ideas: iteration, WHILE statement, DO statement, termination, end-of-data character, endless execution.

One of the major advantages of using a computer is its capacity for repeating a simple procedure over and over with extreme speed. Repetition, often called ***iteration,*** can be used to compute the salaries of 200,000 employees; to carry out complex satellite orbital calculations with 1,000,000 steps; or to search a database of 10,000,000 patients to find a match for a kidney transplant. Since computers can execute millions of statements in a second, and humans cannot possibly write statements that fast, the key to all practical computer applications is iteration, in which statements are written once, but executed many times. Pascal programs provide for the repeated execution of a statement, depending on the outcome of a program condition.

1.5.1 The WHILE Statement

Iterative execution is defined by the Pascal *WHILE statement,* formed using the standard Pascal words `WHILE` and `DO`:

```
WHILE condition
DO
  DO statement
```

If the *condition* is satisfied, the *DO statement* is executed and the WHILE statement is executed again, repeating until (if ever) the condition fails to be satisfied. If the *condition* is not satisfied at the outset, then nothing is done in the WHILE statement. As a rule, a WHILE statement should be written in the style above, with the `WHILE`, `DO` lined up, and the *DO statement* indented by two spaces. Such a standard style makes reading easier and more accurate.

Program 22 uses a WHILE statement to copy any number of characters—tens, hundreds, even thousands or more—from `INPUT` to `OUTPUT`. The number sign character (`#`) is used to mark the end of the input file and stop the copying process.

Program 22: Copy

```
PROGRAM Copy(INPUT,OUTPUT);
  {Transfers all the characters that precede the
   first  #  in INPUT, to OUTPUT.  If  #  is the
   first character in INPUT, a blank line is output.
   The program aborts if INPUT contains no  # .}
  VAR
    Ch: CHAR;
  BEGIN
    READ(Ch);
    WHILE Ch <> '#'
    DO
      BEGIN
        WRITE(Ch);
        READ(Ch)
      END;
    WRITELN
  END.
```

Executions

```
INPUT :ABC#
OUTPUT:ABC

INPUT :Red#AZURE
OUTPUT:Red

INPUT :COPY AZURE#
OUTPUT:COPY AZURE

INPUT :Four score and seven years ago#
OUTPUT:Four score and seven years ago
```

As the second execution shows, any data that follow # are not moved to the output. Clearly, the terminal character cannot be used as data.

Part of the first execution of Program 22 is shown below, with repeated rows corresponding to the iterations of the WHILE statement. The data values are not repeated in the columns of this table where they have not changed. (However, Ch has no value in the last line.)

Partial Execution Table for Program 22

After Execution of	Condition	INPUT	OUTPUT	Ch
BEGIN		**ABC#**	_	?
READ(Ch)		**ABC**		**A**
WHILE Ch<>'#' DO	yes			
BEGIN				
WRITE(Ch)			**A**_	
READ(Ch)		**ABC#**		**B**
END				
WHILE Ch<>'#' DO	yes			
BEGIN				
WRITE(Ch)			**AB**_	
READ(Ch)		**ABC#**		**C**
END				
WHILE Ch<>'#' DO	yes			
BEGIN				
WRITE(Ch)			**ABC**_	
READ(Ch)		**ABC#**_		**#**
END				
WHILE Ch<>'#' DO	no			
WRITELN			**ABC/**_	
END.		**ABC#**	**ABC**	

What would happen if the input for Program 22 did not contain the character **#** ? As the comment claims, the execution will abort. The WHILE repetition would not be stopped by the condition, and eventually all the input would be exhausted, even the extra blank at the end; then the next iteration's READ statement would abort.

1.5.2 Endless Execution

A WHILE statement may go into an *endless execution* if improperly written. For example, if the READ statement inside the WHILE statement of Program 22 were omitted, then the WHILE statement

```
WHILE Ch <> '#'
DO
  BEGIN
    WRITE(Ch)
  END
```

beginning with the value of Ch as **A**, would write an endless sequence of **A**, one for each iteration. While true endless execution is theoretically possible, most Pascal machines would eventually terminate the execution because too much time was used. Endless executions of WHILE statements can use up large amounts of computer time and produce large piles of uninteresting output paper. Therefore, the following is an important rule: The condition used in any WHILE statement should cause the iteration to eventually terminate.

1.5.3 Programs MaxOfAll and CopyOdds

Program 23 determines the maximum character found in INPUT up to the character **#**. The first IF statement writes different messages, depending on whether any data are found in INPUT. The value of the variable Max is a trial maximum—the maximum character found so far in reading INPUT. The concept of setting up a comparison variable often appears in programs. When the execution of the WHILE statement is completed, the value of Max will be the maximum of the data found.

Program 23: MaxOfAll

```
PROGRAM MaxOfAll(INPUT,OUTPUT);
  {Characters from INPUT up to but not including
   the first  #  are considered; the largest such
   character (in the sense of  > ) is the output,
   along with appropriate messages.  The program
   will abort if INPUT contains no  #.}
  VAR
    Ch, Max: CHAR;
  BEGIN
    READ(Ch);
    IF Ch = '#'
    THEN
      WRITELN('NO DATA')
    ELSE
      BEGIN
        WRITE('INPUT DATA IS ');
        Max := Ch;
        WHILE Ch <> '#'
        DO
          BEGIN
            WRITE(Ch);
            IF Ch > Max
            THEN
              Max := Ch;
            READ(Ch)
            END;
        WRITELN;
        WRITELN('MAX OF ALL IS ', Max)
      END
  END.
```

Executions

```
INPUT :AZURE#
OUTPUT:INPUT DATA IS AZURE
       MAX OF ALL IS Z

INPUT :A#
OUTPUT:INPUT DATA IS A
       MAX OF ALL IS A

INPUT :#AZURE
OUTPUT:NO DATA
```

Program 24 copies only every other character from the input to the output. Since it is more complicated than previous programs, it merits more explanatory text in comments, and these appear throughout the program, as a running commentary on what the programmer intends. The program uses variable `Next` to keep track of whether the next character read is odd-numbered, and therefore, whether it should be copied.

Program 24: CopyOdds

```
PROGRAM CopyOdds(INPUT,OUTPUT);
  {Copy every other character of INPUT to OUTPUT,
   up to but not including the first  #.  The
   program will abort if INPUT contains no  #.}
  VAR
    Ch, Next: CHAR;
    {Next is switched between odd (O) and even (E)}
  BEGIN
    Next := 'O'; {odd}
    READ(Ch);
    WHILE CH <> '#'
    DO
      BEGIN
        IF Next = 'O'
        THEN {Copy odd-numbered characters}
          WRITE(Ch);
        READ(Ch);
        {Switch value of Next}
        IF Next = 'O'
        THEN
          Next := 'E' {even}
        ELSE
          Next := 'O' {odd}
      END;
    WRITELN
  END.
```

Executions

```
INPUT :AZURE#
OUTPUT:AUE

INPUT :A#BC
OUTPUT:A

INPUT :123456789#
OUTPUT:13579

INPUT :SEE CAR EAT#
OUTPUT:SECRET
```

The second IF statement within the WHILE statement is:

```
{Switch value of Next}
IF Next = 'O'
THEN
  Next := 'E' {even}
ELSE
  Next := 'O' {odd}
```

Next can only take on the value O or E. If the value is O, it becomes E; if it is E, it becomes O. That is, this IF statement makes Next into a "flip-flop" between O and E.

Two partial execution tables can be used to analyze this IF statement. In the first the value of Next begins as O; in the second it begins as E.

Two Partial Execution Tables for Program 24

After Execution of	Condition	Next
`IF Next = 'O'`	yes	O
`THEN`		
`  Next := 'E'`		**E**
`IF Next = 'O'`	no	**E**
`ELSE`		
`  Next := 'O'`		O

1.5.4 Exercises

1.5.1 Determine the execution tables for the programs listed, using the given input.

a) Program 23, MaxOfAll; INPUT:**BXC#**

b) Program 24, CopyOdds; INPUT:**AZ#**

1.5.2 Break the programs below into lines, indent properly, and state what each does.

a)
```
PROGRAM CopyThirds(INPUT,OUTPUT);
VAR Ch, Next: CHAR; BEGIN Next := '1';
READ(Ch); WRITE('Third chars = ');
WHILE Ch <> '#' DO BEGIN IF Next = '3'
THEN WRITE(Ch); IF Next = '1' THEN
Next := '2' ELSE IF Next = '2' THEN
Next := '3' ELSE {Next = '3'} Next := '1';
READ(Ch) END; WRITELN END.
```

b)
```
PROGRAM MaxDigit(INPUT,OUTPUT); VAR Max,
Ch: CHAR; BEGIN Max := ' '; READ(Ch); WHILE
Ch <> '#' DO BEGIN IF '0' <= Ch THEN IF
Ch <= '9' THEN IF Ch > Max THEN Max :=
Ch; WRITE(Ch); READ(Ch) END; WRITELN;
WRITELN(Max) END.
```

1.5.3 Punctuate the following so that they become programs, place them in proper readable form, and add annotation to the output. What is the output for the given input?

a)
```
PROGRAM More INPUT OUTPUT VAR Ch
CHAR BEGIN READ Ch WHILE Ch <> '#'
DO BEGIN WRITE Ch WRITE ' ' READ
Ch END WRITELN END
```
INPUT:**HELP#**

b)
```
PROGRAM Less INPUT OUTPUT VAR Ch CHAR
BEGIN READ Ch WHILE Ch <> '#' DO BEGIN
IF Ch <> ' ' THEN WRITE Ch READ Ch
END WRITELN END
```
INPUT:**HI OUT THERE#**

c)
```
PROGRAM Medium INPUT OUTPUT VAR Ch1
Ch2 CHAR BEGIN READ Ch1 READ Ch2 WHILE
Ch2 <> Ch1 DO BEGIN WRITE
Ch2 END WRITELN END.
```
INPUT:**Heathrow Airport**

1.5.4 Consider:

```
PROGRAM MysteryLoop(INPUT, OUTPUT);
VAR
  Ch1, Ch2: CHAR;
BEGIN
  READ(Ch2);
  WRITE(Ch2);
  READ(Ch1);
  WHILE Ch1 <= Ch2
  DO
    BEGIN
      WRITE(Ch1);
      Ch2 := Ch1;
      READ(Ch1)
    END;
  WRITELN(Ch1, ' is distinguished.')
END.
```

Describe the behavior of this program when the input is a string of capital letters. In particular, are there strings for which the program fails to terminate? Ones for which it aborts? Add comments to the program to describe exactly what it does.

1.6 Reading CF Pascal Programs

Preview

People must read programs to understand them or make changes to them. Careful choice of variable names, neat formatting, meaningful organization, and sensible comments help the reader of a program to comprehend its function. Programs should be read with great care because even minor changes or typographical errors in the program can dramatically alter the execution.

New ideas: program reading, counting, execution equivalence.

We read programs to discover what they direct the computer to do. We also read programs to learn how to write other programs better. A competent programmer quickly recognizes common patterns and can easily adapt them to novel circumstances. Not all programs are models of excellence. Some programs are too inefficient, some are too arcane, and some are just plain wrong. The Pascal machine simply executes the program as written, not as intended. People must judge the quality of programs. One way to judge a program is to execute it to see if it does the right thing in test cases. But to determine the degree of completeness, correctness, clarity, and efficiency of a program, it is necessary to read and understand the program.

In the following examples, several CF Pascal programs are presented, along with some discussion. After each program, slight changes are suggested. The problem in each case is to determine how the changes will affect the program's behavior. One lesson to be learned from these reading problems is that even a slight change in a program, even a typographical error, can make a radical change in its behavior.

1.6.1 Program DoubleChars

Searching for all adjacent identical characters in the input is a basic pattern that may be useful in some programming tasks. Program 25 `DoubleChars` does this using a read-and-check pattern: it stores the last character read in `Last`, reads the next character into `Next`, then compares the characters to decide whether to write them out.

Program 25: DoubleChars

```
PROGRAM DoubleChars(INPUT,OUTPUT);
  {Input characters preceding  #  are examined,
   and any adjacent pairs are printed.  The
   program aborts if INPUT does not contain  #.}
  VAR
    Last, Next: CHAR;
  BEGIN {DoubleChars}
    READ(Last);
    WHILE Last <> '#'
    DO {Last holds last character read}
      BEGIN
        READ(Next);
        IF Last = Next
        THEN {Print this double char}
          WRITE(Last, Next, ' ');
        Last := Next {Last holds last character read}
      END;
    WRITELN
  END.  {DoubleChars}
```

Executions

```
INPUT :The Mississippi river rolls on and on#
OUTPUT:ss ss pp ll

INPUT :AAA#
OUTPUT:AA AA
```

Reading Problems for Program 25

Reading Problem 1. Suppose that the variables `Last` and `Next` are interchanged everywhere they appear in the entire program. Describe the effect of the changed program.

Reading Problem 2. Suppose that the variables `Last` and `Next` are interchanged just in the DO statement. Describe the effect of the changed program.

Reading Problem 3. Suppose that the variables `Last` and `Next` are interchanged just in the assignment statement. Describe the effect of the changed program.

1.6.2 Program TwoCharacterWords

Printing all two-character words found in the input is another basic pattern. Program 26 (`TwoCharacterWords`) accomplishes this task using a more complicated read-and-check pattern.

Program 26 proceeds through `INPUT` putting each pair of adjacent characters in `Ch1`, `Ch2`, and the character preceding `Ch1`, succeeding `Ch2` in `Last`, `Next` respectively (so the four characters in `Last`, `Ch1`, `Ch2`, `Next` are adjacent in `INPUT`). The scan is initialized by reading the first three characters into `Ch1`, `Ch2`, `Next`, and setting `Last` to a blank (which is required if an initial two-character word is to be found). Then characters from input are moved through the four variables as they are encountered. Each iteration of the scan determines if `Ch1`, `Ch2` constitute a two-letter word, and if so they are printed. This determination is made by the sequence of IF tests

```
IF Last = ' ' THEN
IF Ch1 <> ' ' THEN
IF Ch2 <> ' ' THEN
IF Next = ' ' THEN ...
ELSE IF Next = '#' ...
```

that appear at the beginning of the DO statement. (Notice that the usual indenting would probably be less clear here than aligning all the tests.) The WHILE statement is terminated by Ch2 becoming #, which means that no further two-character words are possible.

Program 26: TwoCharacterWords

```
PROGRAM TwoCharacterWords(INPUT,OUTPUT);
  {All two-character words in INPUT before the
   first  #  are printed.  The program aborts
   if INPUT has no  #.}
  VAR
    Last, Ch1, Ch2, Next: CHAR;
  BEGIN {TwoCharacterWords}
    READ(Ch1);
    IF Ch1 <> '#'
    THEN {INPUT has at least 1 character, now in Ch1}
      BEGIN
        READ(Ch2);
        IF Ch2 <> '#'
        THEN {INPUT has at least 2 characters,
              now in Ch1, Ch2}
          BEGIN
            Last := ' ';
            READ(Next);
            WHILE Ch2 <> '#'
            DO {check if Ch1, Ch2 are a 2-character
                word, and get new characters }
              BEGIN
                IF Last = ' ' THEN
                IF Ch1 <> ' ' THEN
                IF Ch2 <> ' ' THEN
                IF Next = ' '
                THEN {Ch1, Ch2 are 2-char word}
                  WRITE(Ch1, Ch2, ' ')
                ELSE
                  IF Next = '#'
                  THEN {Ch1, Ch2 are 2-char word}
                    WRITE(Ch1, Ch2);
                BEGIN {get new character values}
                  Last := Ch1;
                  Ch1 := Ch2;
                  Ch2 := Next;
                  IF Next <> '#'
                  THEN
                    READ(Next)
                END
              END; {INPUT exhausted}
            WRITELN
          END
      END
  END.  {TwoCharacterWords}
```

Executions

```
INPUT :To be or not to be#
OUTPUT:To be or to be

INPUT :22 is two too#
OUTPUT:22 is
```

Reading Problems for Program 26

Reading Problem 4. Suppose that `Last` and `Next` are interchanged in the WHILE statement. Describe the effect of the changed program.

Reading Problem 5. Suppose that `Ch1` and `Ch2` are interchanged in the WHILE statement. Describe the effect of the changed program.

Reading Problem 6. Suppose that the sequence of four statements in the BEGIN statement with comment `{get new character values}` is reversed (so the IF statement is first, then `Ch2` is set, then `Ch1`, then `Last`). Describe the effect of the changed program.

1.6.3 Program CountChars

Every computer has arithmetic capability, and so does full Pascal. Arithmetic has been omitted from CF Pascal, but the following example illustrates how it can be accomplished using character values.

Program 27 `CountChars` counts the characters of the input up to 999 or notes that there are more than 999 characters. The count is maintained in three variables: `Ones`, `Tens`, and `Hundreds`. The BEGIN statement of the program is a variation of the following simple statement:

```
BEGIN
  READ(Ch);
  WHILE Ch <> '#'
  DO
    READ(Ch)
END
```

These instructions pass over all characters of the input up to the first #, and the number of iterations in the WHILE statement will be the same as the number of characters before #. Thus the characters can be counted by starting `Ones`, `Tens`, `Hundreds` at `0` before the WHILE statement, and advancing them appropriately just after the READ statement within the WHILE statement. First, if `Ones` is less than `9`, only that digit needs to be incremented. Next, if `Ones` is `9`, then it must be set to `0` and `Tens` incremented in the same way. Finally, if `Tens` is `9`, then it must be set to `0` and `Hundreds` incremented. If `Hundreds` is `9`, there are more than 999 characters in the input, and this fact is recorded by setting `Overflow` (which was initialized before the WHILE statement). Because the IF statements to do this work are repetitive and deeply nested, a tabular format is used instead of the usual nested format.

Program 27: CountChars

```
PROGRAM CountChars(INPUT,OUTPUT);
  {The number of characters in INPUT preceding  #
   is printed, if less than 1000; otherwise a
   message is printed.  If INPUT contains no  #,
   the program aborts.}
  VAR
    Ch, Overflow, Ones, Tens, Hundreds: CHAR;
  BEGIN
    Overflow := 'N';
    Ones := '0';
    Tens := '0';
    Hundreds := '0';
    READ(Ch);
    WHILE Ch <> '#'
    DO
      BEGIN
        READ(Ch);
        IF Ones = '0' THEN Ones := '1' ELSE
```

```
          IF Ones = '1' THEN Ones := '2' ELSE
          IF Ones = '2' THEN Ones := '3' ELSE
          IF Ones = '3' THEN Ones := '4' ELSE
          IF Ones = '4' THEN Ones := '5' ELSE
          IF Ones = '5' THEN Ones := '6' ELSE
          IF Ones = '6' THEN Ones := '7' ELSE
          IF Ones = '7' THEN Ones := '8' ELSE
          IF Ones = '8' THEN Ones := '9' ELSE
          IF Ones = '9'
          THEN
            BEGIN
              Ones := '0';
              IF Tens = '0' THEN Tens := '1' ELSE
              IF Tens = '1' THEN Tens := '2' ELSE
              IF Tens = '2' THEN Tens := '3' ELSE
              IF Tens = '3' THEN Tens := '4' ELSE
              IF Tens = '4' THEN Tens := '5' ELSE
              IF Tens = '5' THEN Tens := '6' ELSE
              IF Tens = '6' THEN Tens := '7' ELSE
              IF Tens = '7' THEN Tens := '8' ELSE
              IF Tens = '8' THEN Tens := '9' ELSE
              IF Tens = '9'
              THEN
                BEGIN
                  Tens := '0';
                  IF Hundreds = '0' THEN Hundreds := '1' ELSE
                  IF Hundreds = '1' THEN Hundreds := '2' ELSE
                  IF Hundreds = '2' THEN Hundreds := '3' ELSE
                  IF Hundreds = '3' THEN Hundreds := '4' ELSE
                  IF Hundreds = '4' THEN Hundreds := '5' ELSE
                  IF Hundreds = '5' THEN Hundreds := '6' ELSE
                  IF Hundreds = '6' THEN Hundreds := '7' ELSE
                  IF Hundreds = '7' THEN Hundreds := '8' ELSE
                  IF Hundreds = '8' THEN Hundreds := '9' ELSE
                  IF Hundreds = '9'
                  THEN Overflow := 'Y'
                END
            END
        END;
      IF Overflow = 'N'
      THEN
        WRITELN('Number of characters is ',
                  Hundreds, Tens, Ones)
      ELSE
        WRITELN('Number of characters exceeds 999')
    END.
```

Execution

```
INPUT :There are forty characters in this file.#
OUTPUT:Number of characters is 040
```

Reading Problems for Program 27

Reading Problem 7. Suppose that the variables Ones and Hundreds are interchanged throughout the WHILE statement. Describe the effect of the changed program.

Reading Problem 8. Suppose that each ELSE word within the WHILE statement were replaced by a semicolon. Describe the effect of the changed program.

Reading Problem 9. Suppose that the assignments of 0 to Ones and Tens within the WHILE statement are deleted. Describe the effect of the changed program.

1.6.4 Execution-Equivalent Programs

Different programs can lead to the same results in execution. It is not surprising that two programs with similar statements and structure produce identical results. But it may be a surprise to discover that very unlike programs can also produce identical results. For example, the following three statements cannot be distinguished in execution:

```
BEGIN
  Ch1 := Ch2;
  Ch2 := Ch1
END
```

```
IF Ch1 <> Ch2
THEN
  Ch1 := Ch2
```

```
WHILE Ch1 <> Ch2
DO
  Ch1 := Ch2
```

In each case, the result is the same as that produced by:

```
Ch1 := Ch2
```

In the BEGIN statement, the second assignment is superfluous, since Ch1 is set to Ch2 in the first assignment and Ch2 is then set to its own value in the second. In the IF statement, if the values in Ch1 and Ch2 are the same, there is no need to make an assignment, while if they differ, the assignment is made in the THEN part. In the WHILE statement, the assignment is made only if necessary, and the iteration terminates as soon as the assignment has been made.

The execution steps are different in each case, but the results are the same. In the IF statement, the condition will be evaluated once; in the WHILE statement, the condition will be evaluated once or twice. Indeed, the unnecessary complexity in these examples can be repeated and nested with larger statements that take more execution steps to make this simple assignment. One such example is:

```
BEGIN
  WHILE Ch1 <> Ch2
  DO
    BEGIN
      Ch1 := Ch2;
      Ch2 := Ch1
    END;
  IF Ch1 <> Ch2
  THEN
    Ch1 := Ch2
END
```

No one would write this statement to do a simple assignment, but sometimes excessively complex programs emerge from poor design or incompetent modification. It is very useful to learn how to reduce complex statements to a simpler form with identical execution results. This knowledge can be applied in reworking programs to increase readability, maintainability, and performance.

1.6.5 Exercises

1.6.1 The following program TwoCharacterWords2 is intended to have the same behavior as Program 26, TwoCharacterWords, but does not. What does the program do wrong? Can you isolate part of the program as being at fault?

```
PROGRAM TwoCharacterWords2(INPUT,OUTPUT);
  VAR
    Ch1, Ch2, Next: CHAR;
  BEGIN {TwoCharacterWords2}
    Next := ' ';
    READ(Ch1);
    WHILE Ch1 = ' '
    DO
      READ(Ch1);
      {Ch1 has the first nonblank}
    WHILE Ch1 <> '#'
    DO
      BEGIN {Check this word for being
             two characters long}
        READ(Ch2);
        IF Ch2 <> ' '
        THEN
          IF Ch2 <> '#'
          THEN {Word is at least two char long}
            BEGIN {Check if longer}
              READ(Next);
              IF Next = ' '
              THEN {Word is exactly two long}
                WRITE(Ch1, Ch2, ' ')
              ELSE {Next is not a blank}
                IF Next = '#'
                THEN {Word is two long}
                  WRITE(Ch1, Ch2, ' ')
                ELSE {Word is longer than two}
                  WHILE Next <> ' '
                  DO
                    READ(Next);
              Ch2 := Next
            END;
        Ch1 := Ch2;
        WHILE Ch1 = ' '
        DO
          READ(Ch1)
      END; {Handling a word}
    WRITELN
  END.  {TwoCharacterWords2}
```

1.6.2 Modify Program 26, TwoCharacterWords, so that it finds three-character words instead.

1.6.3 Program 27 CountChars has two READ statements, one inside the WHILE statement, and one before it. But the counting is done only within the WHILE statement. Why isn't the count off by 1?

1.6.4 Describe the effect on Program 27 CountChars of omitting each of the following portions of the program.

a) Initialization of `OverFlow` to `N`.

b) The first READ statement.

c) The second READ statement.

1.6.5 In the IF statement of Program 21 `Max3`, suppose that `Ch3` is replaced by `Ch1`. Write the resulting statement, and simplify it. What does the program do with this change?

1.6.6 In Program 23 `MaxOfAll`, replace `Max` by `Ch` everywhere. What does the new program do?

1.6.7 Let B be a condition, and S and T be statements that are different. In any program, can the code fragment

```
IF B
THEN
  S;
IF B
THEN
  T
```

always be replaced by the fragment

```
IF B
THEN
  BEGIN
    S;
    T
  END
```

with the same results? Explain. No — if S changes B

1.7 Chapter Summary

The programming language CF Pascal has been described in outline, and enough sample programs have been given to show something of how programs act when executed by the Pascal machine. Programs have a rigid form, in which statements appear in sequence for execution. Variables are names for values that may change during execution; the list of variables precedes the statements using those variables. CF Pascal uses only character data—the input is a string of characters, as is the output, and internal values are single characters. Statements come in several forms: WRITE statements to print text and the values of variables; READ statements to obtain variable values from the input; IF statements that can select which group of statements is done next; WHILE statements that allow arbitrary repetition of a statement group. BEGIN statements group statements together and can be particularly useful within IF and WHILE statements.

The most important idea introduced in this chapter is the execution table. An execution table summarizes the actions of the Pascal machine executing a given program in a clear, step-by-step fashion. The rows of the table show the statements executed, and the columns show the values of variables, input, and output after each statement is executed. The table describes execution for only a single input, but it is invaluable in understanding what a program does.

Programs may execute normally, if statements are executed in sequence through the program as IF and WHILE statements dictate, and the final `END` is reached. But execution may also abort, if an attempt is made to read input that does not exist. Execution may also be endless, if the condition tested in a WHILE statement remains true forever.

Programming is often a matter of recognizing the need for part of a program that has been written before, and reusing it. Several such ideas were introduced in this chapter: interchanging values of variables; copying values from input to output; echoing input and annotating output; finding the maximum of a set of values as they are obtained; isolating patterns in input; counting using character values.

CHAPTER 2

WRITING PROGRAMS IN CF PASCAL

Chapter Preview

In Chapter 2, you will learn how to write simple programs in CF Pascal, verify correct syntax, and use principles of program design and development in a process of stepwise refinement to write, execute, and test simple programs.

2.1 An Introduction to CF Pascal Syntax

Preview

CF Pascal programs are formed with an exact syntax. A precise set of rules defines this syntax, and procedures for verifying correct syntax are given.

New Ideas: program syntax, BNF notation, syntax rule, alternative syntax rule, recursive syntax rule, syntax tree, syntax correctness proof.

The first need in writing English is to master the syntax, to be able to write complete sentences, form them into paragraphs, and so on. The syntax of English sentences takes many years to master because English is a natural language and subject to innovation and modification in general usage.

In the same way, the first need in writing CF Pascal is to master the syntax, to be able to write complete statements and form them into programs. However, CF Pascal is a formal language with a precise set of rules for its syntax. These rules permit the syntax of CF Pascal to be mastered more quickly and completely than is possible with a natural language. Once the syntax is mastered, you can turn to the question of what programs to write to solve problems at hand.

The Pascal machine treats a Pascal program as a sequence of characters. By contrast, people recognize meaningful words and might guess what is intended in the following character string that is not a Pascal program:

```
PROGRAM CopyChar(INPUT,OUTPUT);
  VAR
    Ch; CHAR;
  BEGIN
    READ(Ch);
    WRITELN(Ch)
  END.
```

The Pascal program with this name that appeared in Chapter 1 is:

```
PROGRAM CopyChar(INPUT,OUTPUT);
  VAR
    Ch: CHAR;
  BEGIN
    READ(Ch);
    WRITELN(Ch)
  END.
```

The single-character difference between these character strings (a colon/semicolon in the CHAR declaration) will cause the Pascal compiler to reject the first string.

Because of the need for absolute character-by-character precision in forming Pascal programs, this chapter introduces exact syntax rules for CF Pascal. Examples and experimentation are useful during learning, but precise and concise rules allow you to verify with confidence the syntactic correctness of programs. Learning Pascal only by examples and using the Pascal machine to confirm guesses about correct syntax enforces not only a limited use of Pascal but also some wrong ideas about writing programs.

2.1.1 Program Syntax

The `CopyChar` program is a special instance of a character string that conforms to the Pascal syntax rules. These rules of syntax define parts of programs (much like parts of speech in English) by defining syntax categories, and showing how parts of programs fall into those categories.

Syntax parts can be specified by the rules of grammar in *BNF (Backus-Naur Form)* notation. (John Backus and Peter Naur are computer scientists who, along with others, collaborated on an important programming language called ALGOL 60—a conceptual forerunner of Pascal.) A syntax part is denoted by a word or phrase within angle brackets (<...>) and is defined by a rule that may refer to other syntax parts.

Syntax Rule 1 (SR1) defines a syntax part called "program."

SR1. <program> ::= <heading> ; <block> .

Syntax Rule 1 states that any syntax part named <program> must be formed as a syntax part named <heading>, followed by a semicolon, followed by a syntax part called <block>, followed by a period. The angle brackets enclosing words signify that these are the names of syntax parts each defined by a syntax rule. This rule defines <program>, while <heading> and <block> will be defined by other rules. The semicolon and the period are syntax literals, or character strings that must appear exactly as shown in a Pascal program.

The three-character symbol ::= means "is any instance of." Syntax Rule 1 is read as:

A <program> is any instance of a <heading>, followed by a semicolon, followed by a <block>, followed by a period.

The symbol ::= and the angle brackets < > are called *metasymbols* because they are not part of the language being defined. Neither ::= nor <program> will appear in a Pascal program except possibly in a comment. In the case of `CopyChar`, the character string

```
PROGRAM CopyChar(INPUT,OUTPUT)
```

corresponds to the syntax part <heading>, and the character string

```
VAR
  Ch: CHAR;
BEGIN
  READ(Ch);
  WRITELN(Ch)
END
```

to the syntax part <block>. In order to form the whole program to correspond to the syntax part <program>, a semicolon needs to be inserted just after this <heading>, and a period must be added after this <block>, as prescribed by Syntax Rule 1. We talk about parts of programs by saying which character string corresponds to which syntax part, and then referring to that string by the name of the part. The angle brackets help to avoid confusion when the names within them are English words.

Next, Syntax Rule 2 defines the syntax part <heading>.

SR2. <heading> ::= `PROGRAM` <identifier> `(INPUT,OUTPUT)`

A <heading> begins with the literal character string `PROGRAM`, followed by an <identifier>, then by the literal character string `(INPUT,OUTPUT)`. In the program `CopyChar`, this <identifier> is the character string `CopyChar` itself.

Standard words such as PROGRAM and <identifier>s must be separated from each other by at least one blank (but there may be more than one) unless there is a punctuation character such as a semicolon already separating them. In the syntax rules themselves blank characters are never shown explicitly. Blanks are used in the syntax rules just to make them more readable. Where one blank may be inserted, any number may be; a line break may replace any blank.

An <identifier> is any character string that begins with a letter and is followed by any number (zero or more) of letters and/or digits, with no intervening blanks or other characters. Syntax Rule 3 gives the definition, using two new ideas. First, Syntax Rule 3 uses a vertical bar ¦ whose meaning is "or." Second, this rule uses the very syntax part being defined (in this case, <identifier>) in its own definition. The vertical bar is a another metasymbol for giving rules of syntax.

SR3. <identifier> ::= <letter>
¦ <identifier> <letter>
¦ <identifier> <digit>

This is read:

An identifier is a letter, or an(other) identifier followed by a letter, or an(other) identifier followed by a digit.

Each possibility in such a rule is called an *alternative* of the rule. Should it be necessary to refer to one of the alternatives, a Dewey decimal notation will be used. For example, SR3 has first alternative 3.1 (<letter>), and its other two alternatives are 3.2 and 3.3. In brief, Syntax Rule 3 states that any letter or digit added to the end of an <identifier> makes a new <identifier>. That is, an <identifier> of one letter is defined by 3.1, while 3.2 and 3.3 define <identifier>s of more than one character in terms of <identifier>s with one less character.

Syntax Rule 3 is called a *recursive definition* because <identifier> is used in its own definition. No actual character string that is an <identifier> is used to define itself; some <identifier>s are used to define others (one character longer). There is no limit to the length of an <identifier>.

The syntax parts <letter> and <digit> are defined to mean what they seem, namely:

SR4. <letter> ::= A ¦ B ¦ C ¦ D ¦ E ¦ F ¦ G ¦ H ¦ I ¦ J ¦ K ¦ L ¦ M ¦ N ¦ O ¦ P ¦ Q ¦ R ¦ S ¦ T ¦ U ¦ V ¦ W ¦ X ¦ Y ¦ Z ¦ a ¦ b ¦ c ¦ d ¦ e ¦ f ¦ g ¦ h ¦ i ¦ j ¦ k ¦ l ¦ m ¦ n ¦ o ¦ p ¦ q ¦ r ¦ s ¦ t ¦ u ¦ v ¦ w ¦ x ¦ y ¦ z

SR5. <digit> ::= 0 ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 ¦ 8 ¦ 9

The alternative that allows a <letter> to be G (for example) is named 4.7. Now, using these definitions, we can build up the <identifier> CopyChar a character at a time:

<identifier>	By Syntax Rules
C	C is a <letter> by 4.3, hence an <identifier> by 3.1
Co	C is an <identifier> as above, o is a <letter> by 4.41, so Co is an <identifier> by 3.2
Cop	Similarly, by Co above and 4.42 and 3.2
...	...
CopyChar	Similarly, by CopyCha above and 4.44 and 3.2

Each step can be traced back to the syntax rules. Similarly, character strings Copy1, XxXxX, One4Two are <identifier>s.

Because blanks appear in syntax rules only to improve readability, a separate rule must be given for them: no blanks may appear within an <identifier>.

The following are *not* <identifier>s:

Characters	Not an <identifier> because
2nd	does not start with a letter
C 75	contains a blank
Ch1+Ch2	the plus sign is not a letter or a digit

Identifiers appear so often in programs that their recognition and construction should become automatic.

2.1.2 Syntax Trees

The syntax rules of CF Pascal define a step-by-step construction of any program as a string of characters. Conversely, if there is no such construction of the character string, the string cannot be the text of a Pascal program. The construction of a program can be shown as a *syntax tree*. Figure 1 gives part of the construction for the <program> CopyChar.

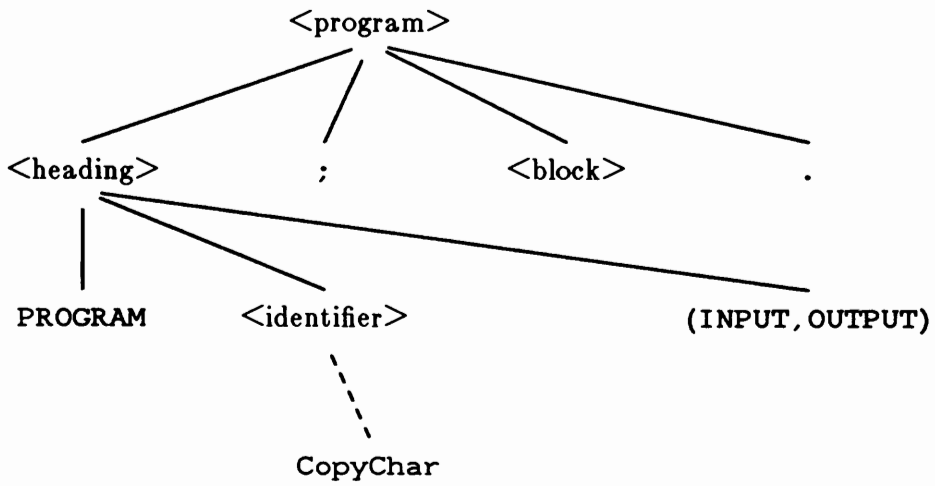


Figure 1: Partial Syntax Tree for CopyChar

At the top (called the *root* of the tree—it "grows" upside down!) is <program>, and attached to it lower down are the four parts from SR1: <heading>, the semicolon, <block>, and the period. Then hanging from <heading> are its three parts from SR2. The tree does not show the details of the construction of <identifier>, but the result of that construction is shown attached with a dashed line, to indicate that the details have been abbreviated. The final result can be obtained by assembling the characters at the lowest ends of lines (they are called the *leaves* of the tree) from left to right, where there appear in order:

PROGRAM CopyChar(INPUT,OUTPUT); <block> .

This is thus the <program>, complete except for its <block>, whose syntax rule has not yet been given, and so cannot be filled out yet. A blank has been inserted between PROGRAM and CopyChar according to the separate rule for separating standard words and <identifier>s; but, no blank was needed between CopyChar and INPUT, since the punctuation character intervened; blanks are not allowed within CopyChar.

Figure 2 gives a full construction of the <identifier> Copy1.

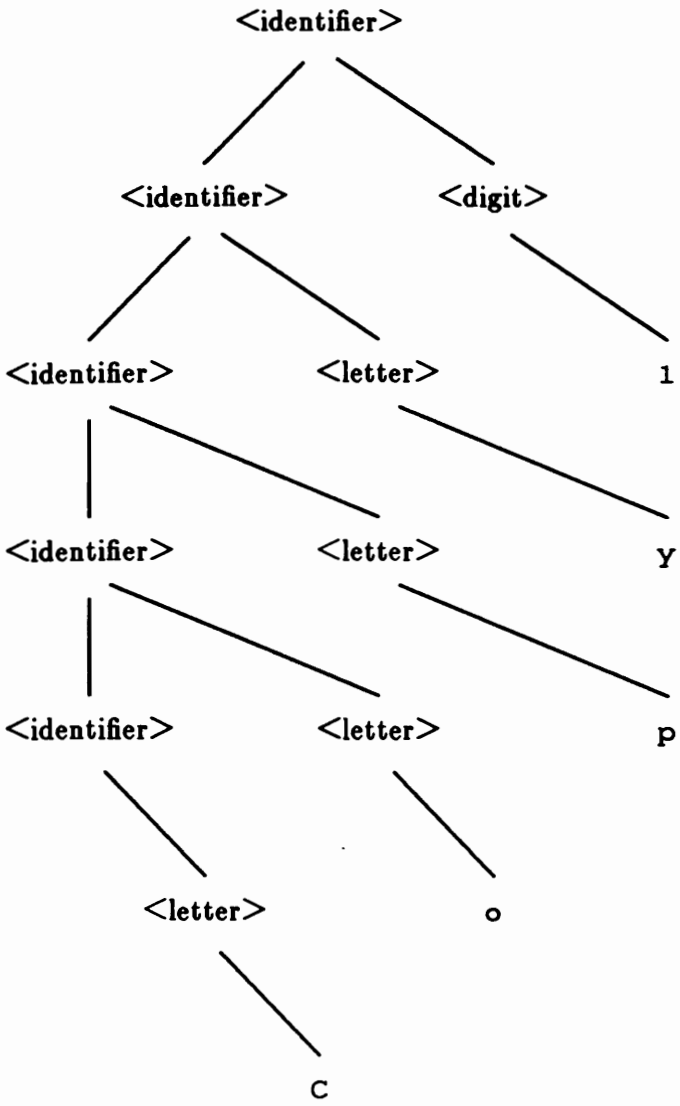


Figure 2: Syntax Tree for <identifier> Copy1

When a syntax tree is constructed using all the necessary syntax rules, the leaves of the tree must all actually appear in the program. When these are read from left to right around the tree, inserting blanks where required, they spell out the program part whose name in angle brackets appears at the root of the tree. For example, part of Figure 1 is reproduced as Figure 3, showing that a tree rooted at <heading> can construct

```
PROGRAM CopyChar(INPUT,OUTPUT)
```

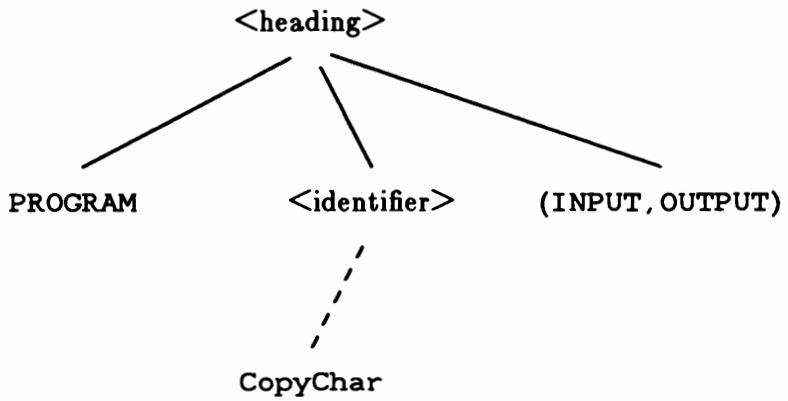


Figure 3: Syntax Tree for a <heading>

For short we say that

```
PROGRAM CopyChar (INPUT, OUTPUT)
```

is a <heading>. Then when we talk about "the <heading>," it means this string. But often the particular string will not matter, and a reference to "a <heading>" means *any* string that could be so constructed. (In this example, the only thing that could change in the tree would be the <identifier>.)

Syntax trees can be used to prove the syntax correctness of any character string that is a program part. Starting with a syntax rule having this part on the left side of ::=, the tree is constructed from this and other syntax rules. The syntax of a character string is proved correct if it is formed from the leaves of the tree by inserting blanks appropriately. The general blank insertion rule is:

1. No blanks may be inserted in an <identifier>.
2. A blank must be inserted between standard words and identifiers, unless a punctuation character already intervenes. A blank may be inserted before or after standard words, <identifier>s, and punctuation characters.
3. Where one blank can be inserted, any number can be.
4. A line break may appear anywhere a blank can; however, a line break does not replace a blank for the purpose of 2. above.

For example, the <heading> tree above shows that (among others)

```
PROGRAM      CopyChar  (INPUT, OUTPUT)
```

is a <heading>, and so is:

```
PROGRAM
  CopyChar
    (INPUT, OUTPUT)
```

showing different choices of blanks and line breaks. In some Pascal machines a line break is literally ignored; in others it does replace a blank. Rule 4. is a conservative one that is safe under all circumstances.

As a check on the proper construction of a syntax tree, the numbers of the syntax rules used in building it can be attached to each node, as shown for the <identifier> B4U in Figure 4.

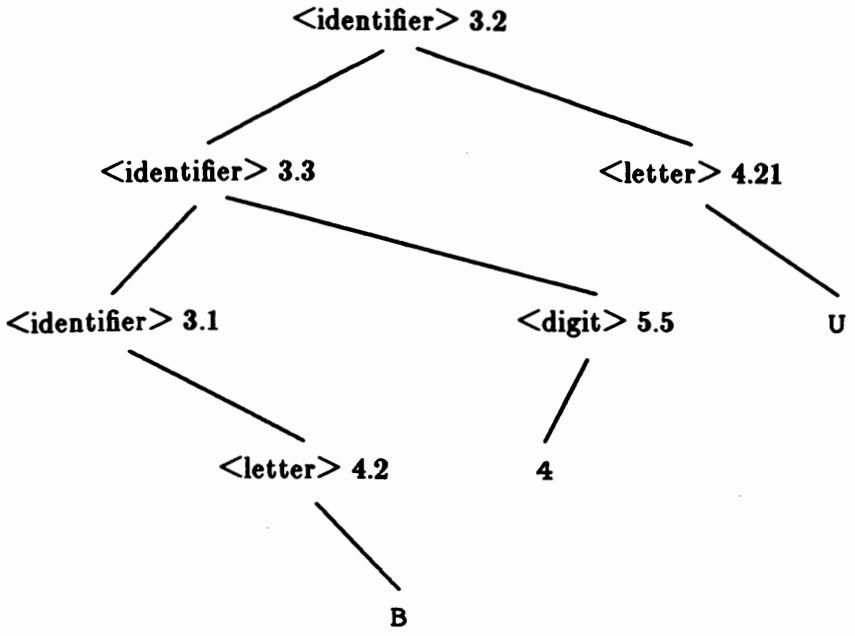


Figure 4: Syntax Tree for <identifier> B4U

For example, the rule used at the root is the 2nd alternative of SR3, so it is marked 3.2; the rule used to get the literal character U is the 21st alternative of SR4, so marked 4.21.

With experience, these syntax proofs become easy and automatic. Once the idea is well understood, it will seldom be necessary to draw a syntax tree. But until BNF becomes second nature, the definition of syntax correctness requires drawing the tree.

2.1.3 Block Syntax

The syntax of <block> is given in Syntax Rule 6:

SR6. <block> ::= <declarations> ; <BEGIN statement>
¦ <BEGIN statement>

and the block of PROGRAM CopyChar will turn out to be an instance of 6.1. The syntax of <declarations> is given in Syntax Rules 7 and 8:

SR7. <declarations> ::= VAR <identifier list> : CHAR

SR8. <identifier list> ::= <identifier>
¦ <identifier list> , <identifier>

Syntax Rule 8 gives a recursive definition for <identifier list>, which is any list of one or more identifiers, separated by commas.

In addition to the syntax rules, CF Pascal programs must satisfy the following *context rules:*

CR1. All identifiers in a declaration's identifier list must be unique.

CR2. No reserved word, e.g. IF, WHILE, etc. can be used as an identifier. (A list of the reserved words in Pascal is given in Appendix II.)

Figure 5 shows that

```
VAR Ch: CHAR
```

is a <declarations>.

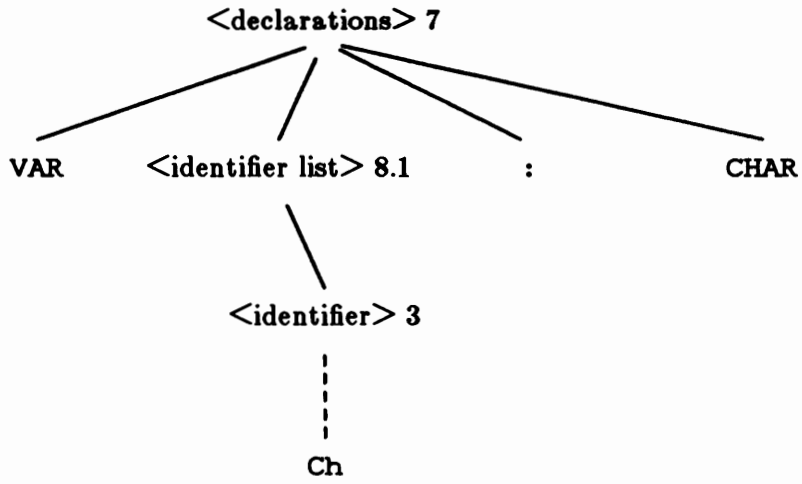


Figure 5: Syntax Tree for a <declarations>

The style suggested in Chapter 1 for this <declarations> is

```
VAR
  Ch: CHAR
```

illustrating the rule about blanks and line breaks.

The syntax of <BEGIN statement> is given in Syntax Rules 9 and 10, and Context Rule 3 applies, as well.

SR9. <BEGIN statement> ::= BEGIN <statement list> END

SR10. <statement list> ::= <statement>
| <statement list> ; <statement>

CR3. Any <identifier> in a <statement>, other than standard identifiers e.g. READ, WRITE, etc., must appear in the <identifier list> of the <declarations>. (The standard identifiers are listed in Appendix II.)

Syntax Rule 10 gives a recursive definition for <statement list>, which is any sequence of <statement>s, separated by semicolons. Note that no semicolon is required after the last statement, just before the END word. An example of a two-statement <BEGIN statement> is shown in Figure 6.

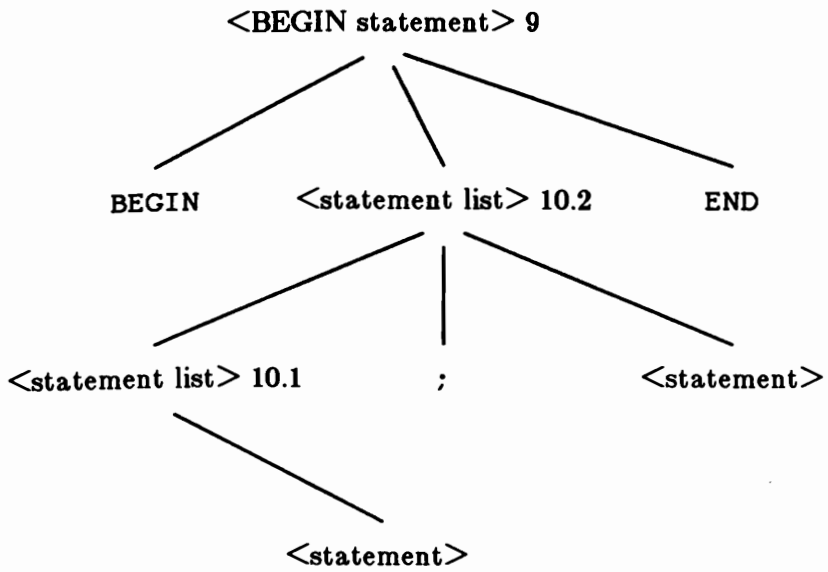


Figure 6: Syntax Tree for a <BEGIN statement>

2.1.4 Exercises

2.1.1 In each part below, does the definition given agree with SR3 for <identifier>? Explain why or why not.

a) <identifier> ::= <letter>
¦ <letter> <identifier>
¦ <digit> <identifier>

b) <identifier> ::= <letter>
¦ <letter> <identifier>
¦ <identifier> <digit>

c) <identifier> ::= <letter>
¦ <identifier> <next>
<next> ::= <digit> ¦ <letter>

d) <identifier> ::= <letter>
¦ <identifier> <rest>
<rest> ::= <letter> ¦ <digit>
¦ <letter> <rest>
¦ <digit> <rest>

2.1.2 Given the following rules, and the convention that no blanks can be inserted between the leaves, what do the character strings defined by <both ends> look like? Describe them in English.

<both ends> ::= <letter>
¦ <letter> <both ends>
¦ <both ends> <digit>

2.1.3 Write a definition for <ident4max> in BNF that constructs identifiers consisting only of <letter>s, which are of length one to four characters, and those identifiers only.

2.1.4 Verify that

```
PROGRAM A1(INPUT,OUTPUT)
```

is a <heading> by constructing a syntax tree.

2.1.5 The following character strings are not <heading>s. By attempting to construct a syntax tree, show where each one fails.

```
PROGRAM A1(INPUT;OUTPUT)
PROGRAM A1,Also(INPUT,OUTPUT)
PROGRAM, A35(INPUT,OUTPUT)
```

2.1.6 Rules 1 through 8 are a grammar for automobile descriptions, starting with <cars>:

Rule 1. <cars> ::= <basics> <paint>
¦ <basics> <two-tone>

Rule 2. <basics> ::= <body> <engine> <transmission>

Rule 3. <body> ::= four-door
¦ <two-door options>

Rule 4. <two-door options> ::= two-door
¦ two-door hatchback

Rule 5. <engine> ::= four cylinder
¦ six cylinder

Rule 6. <transmission> ::= automatic
¦ four-speed manual
¦ five-speed manual

Rule 7. <paint> ::= red ¦ blue ¦ white ¦ silver ¦ green

Rule 8. <two-tone> ::= <paint> and <paint>

One blank must occur between each item taken from the leaf of a syntax tree to form sentences using this grammar.

a) Write out an exhaustive list of all the sentences that are <basics>, that is, all the sentences that can be constructed from syntax trees rooted at <basics>.

b) Now considering all eight rules, how many correct <cars> are there?

c) Find and write the <cars> with the fewest characters.

d) Find and write the <cars> with the most characters.

e) Which sentence has the shortest syntax tree? Give the tree.

f) Which of a) - e) would change if the rule about blanks were changed to allow any number of blanks between leaves?

2.1.7 Write syntax rules needed to define a mailing address that consists of a name, an address, and a place. The name consists of a first name, middle initial with a period or no middle initial at all, and a last name. The address consists of a number and a street name. The place consists of a city, followed by a comma, a two-letter abbreviation, and a five-digit zip code. What must be added to these rules before it is possible to construct any mailing addresses?

2.1.8 Write syntax rules to define valid telephone numbers (including area codes) in the form

(ABC)CDD-DDDD

where A is in the range 2 through 9; B is either 0 or 1; C is in the range 1 through 9; and D is in the range 0 through 9. What should be the convention about placing blanks when items are taken from the syntax trees?

2.1.9 Suppose that two different syntax rules allow the construction of exactly the same character strings. Do the two syntax trees for the same string have to be the same using the different rules? Explain.

2.2 The Syntax of CF Pascal Statements

Preview

The syntax of <statement> is given, along with the auxiliary definitions needed to define each kind of statement. Abbreviation in syntax trees is a powerful way to tailor a proof of syntax correctness to the level of sophistication of the reader. Comments are inserted into programs after the tree is constructed.

It will complete the definition of <block> and of <program> to define <statement>:

SR11. <statement> ::= <READ statement>
¦ <WRITE statement>
¦ <assignment statement>
¦ <null statement>
¦ <BEGIN statement>
¦ <IF statement>
¦ <WHILE statement>

The syntax of READ and WRITE statements is:

SR12. <READ statement> ::= READ (<identifier list>)

SR13. <WRITE statement> ::= WRITE (<write list>)
¦ WRITELN (<write list>)
¦ WRITELN

SR14. <write list> ::= <write item>
¦ <write list> , <write item>

SR15. <write item> ::= <identifier>
¦ ' <character string> '

SR16. <character string> ::= <character>
¦ <character string> <character>

The definitions for <write list> and <character string> are recursive, using the same pattern as <identifier list> given above. In a <write list> there is a comma separator; in a <character string> there is no separator, and no blanks may be inserted in a <character string>. However, we do want blanks as <character>s, so to show them the special symbol □ will be used.

SR17. <character> ::= <letter> ¦ <digit> ¦ <special character>

SR18. <special character> ::= (¦) ¦ = ¦ { ¦ } ¦ : ¦ ; ¦ ''
¦ □ ¦ < ¦ > ¦ , ¦ . ¦ #

Some Pascal machines have a larger set of <special character>s, but these are all we will need for CF Pascal.

There are now enough definitions to establish that

WRITELN (Ch)

is a <statement>, as shown in Figure 7.

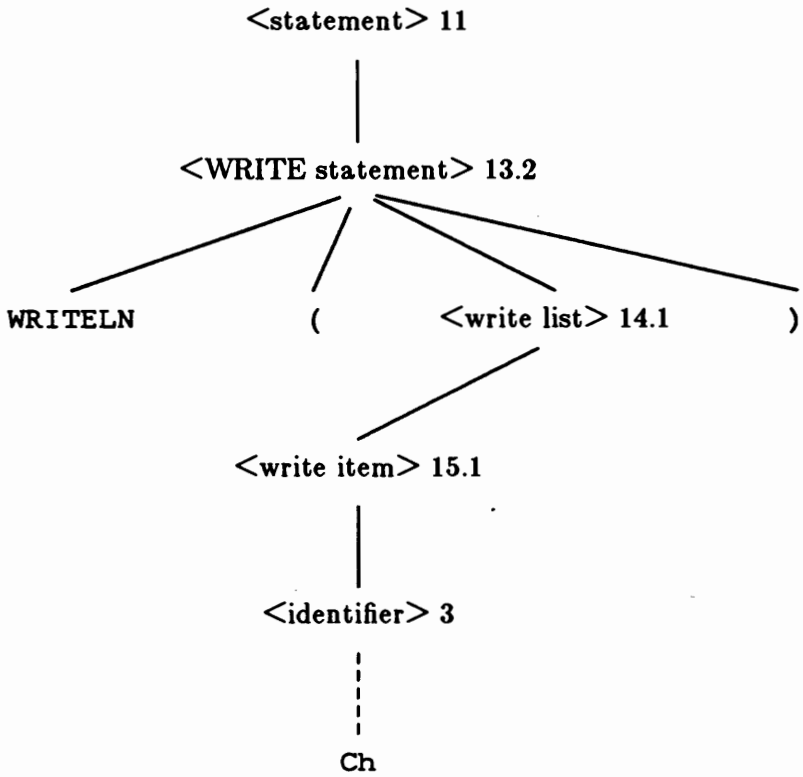


Figure 7: Syntax Tree for a WRITELN statement

All the rules for a full syntax-tree construction of the simple program `CopyChar` have now been given. Figure 8 abbreviates the construction.

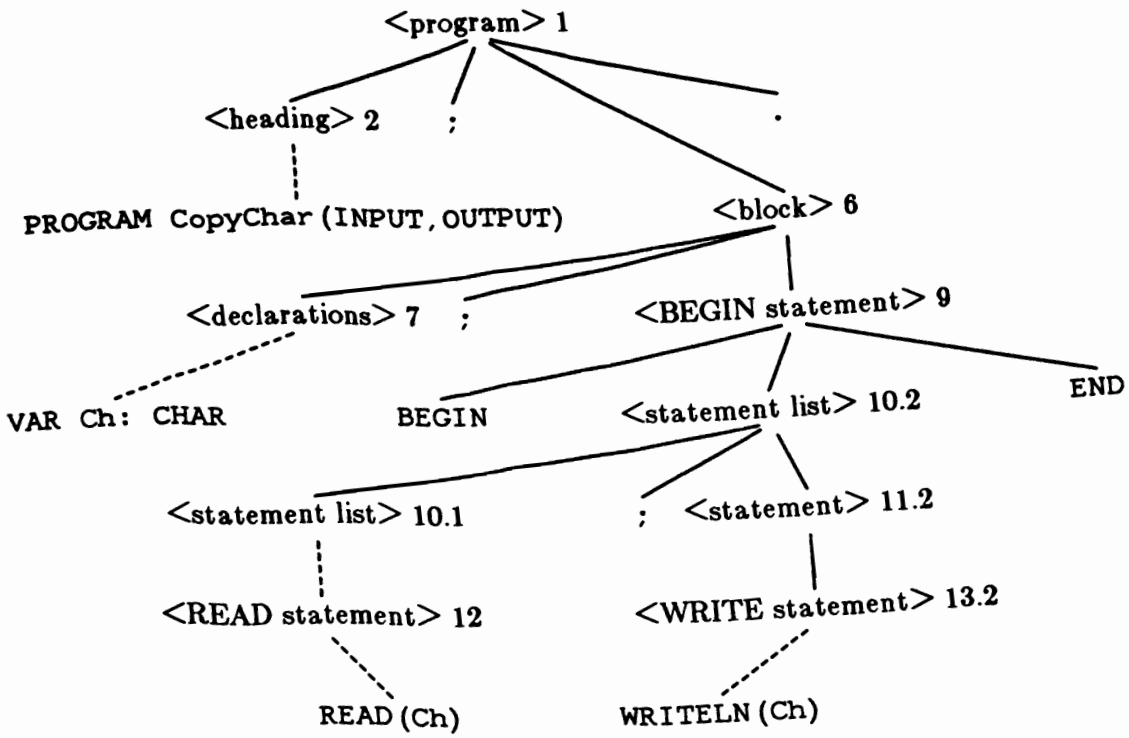


Figure 8: Abbreviated Syntax Tree for CopyChar

The abbreviations are shown with dashed lines. For example, the expansion of the <READ statement> is given in Figure 9.

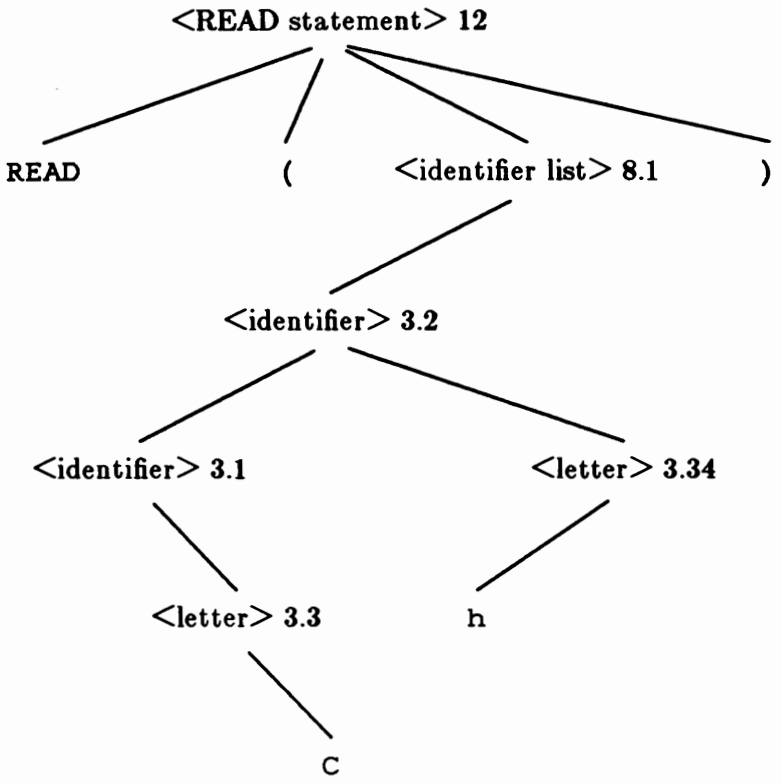


Figure 9: Expansion of Part of Figure 8

There are no rigid rules for abbreviating in syntax trees. The purpose of the tree is to prove that a certain character string has correct syntax. If that proof is clear using abbreviations, then they should be used.

The syntax tree of Figure 8 summarizes the proof that

```
PROGRAM CopyChar(INPUT,OUTPUT);
  VAR
    Ch: CHAR;
  BEGIN
    READ(Ch);
    WRITELN(Ch)
  END.
```

is a <program>. The proof consists of two parts: (1) checking that the tree is correctly formed, that each syntax rule is correctly used to obtain the level beneath it; and (2) checking that the leaves when assembled from the left, with blanks (or line breaks) inserted properly, do form this character string.

Even the abbreviated tree for CopyChar has 19 lines, which may seem excessive effort to prove that such a small program is syntactically correct. However, CopyChar requires a minimum of 72 characters, and a single character change can destroy its syntax correctness. So 19 lines may not be excessive after all. The difference between using a syntax tree, and comparing a program to a set of examples, is the difference between a powerful, complete method and guesswork. The examples may work in simple cases, but the method is needed to be sure when things get complex.

Continuing with the syntax of CF Pascal <statement>s:

SR19. <assignment statement> ::= <identifier> := <expression>

SR20. <expression> ::= <identifier>
¦ ' <character> '

For example,

```
Ch1 := Ch2;
Ch1 := 'A'
```

is a <statement list> composed of two <assignment statement>s, but the following are not <assignment statement>s:

Text	Reason
`'A' := Ch2`	left side is not an <identifier>
`Ch1 := 'ABC'`	right side is not an <expression> because ABC is not a <character>

SR21. <null statement> ::=

The right side of this rule is missing; thus, a null statement contains no characters at all.

SR22. <IF statement> ::= IF <condition> THEN <statement>
¦ IF <condition> THEN <statement> ELSE <statement>

SR23. <condition> ::= <expression> <comparison> <expression>

SR24. <comparison> ::= < ¦ = ¦ > ¦ <= ¦ <> ¦ >=

In rule 24, notice that the symbols for less than, etc., are not the metasymbols < >, but <special character>s.

For example,

```
IF Ch1 < Ch2 THEN Ch1 := Ch2
```

and

```
IF Ch1 < 'A'
THEN
  Ch2 := 'B'
```

and

```
IF 'A' > 'B'
THEN {nothing}
ELSE
  Ch1 := Ch2
```

are all <IF statement>s, but:

```
IF 'A' > 'B'
THEN {nothing};
ELSE
  Ch1 := Ch2
```

is not an <IF statement> because it has an improper internal semicolon.

Finally, here is the syntax of the WHILE statement:

SR25. <WHILE statement> ::= WHILE <condition> DO <statement>

For example,

```
WHILE Ch1 <> Ch2 DO Ch2 := Ch1
```

and

```
WHILE 'A' < 'B'
DO
  Ch1 := 'A'
```

are <WHILE statement>s, but

```
WHILE Ch1 <> Ch2 DO; Ch2 := Ch1
```

is not a <WHILE statement> because of an improperly placed semicolon. In fact, the last example *is* a <statement list>, as Figure 10 shows.

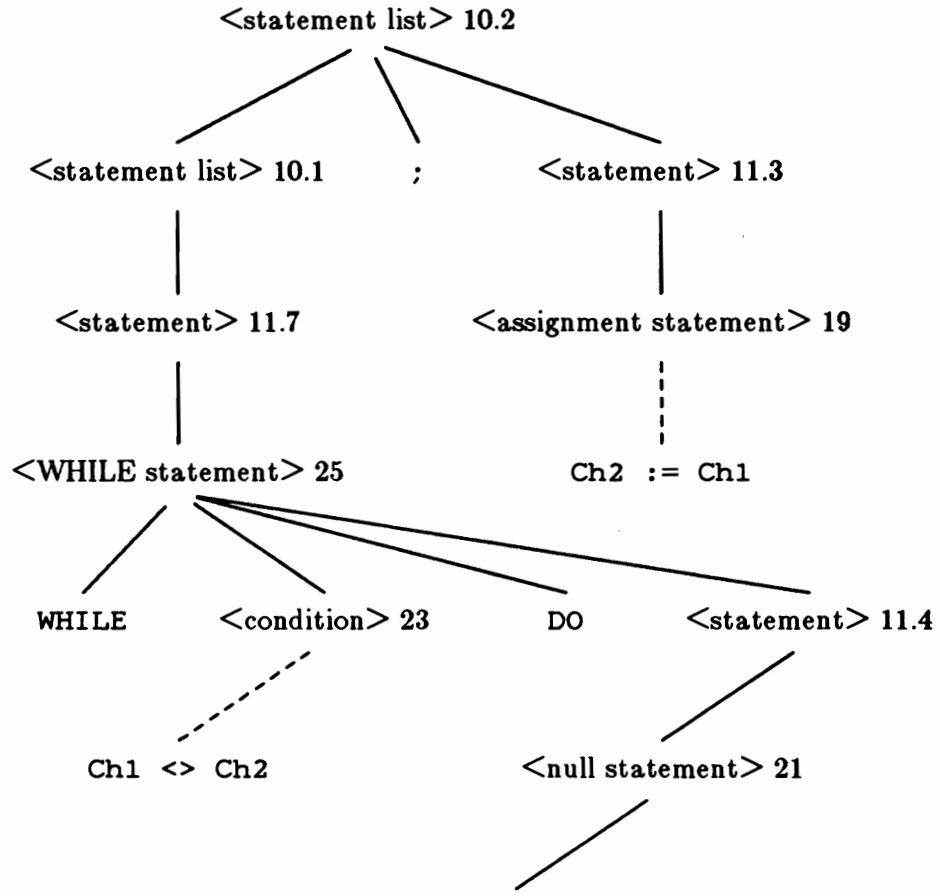


Figure 10: Syntax Tree for an Unusual <statement list>

If the programmer thought this was a <WHILE statement>, perhaps the <assignment statement> was to cause the <condition> to fail and the iteration to cease. But in fact the <statement list> that has actually been written will produce endless execution should it happen that Ch1 and Ch2 do not have the same value. The worst thing that can happen as a result of a mistake in syntax is not an error message!

Figure 11 summarizes a proof that

```
IF Ch1 < 'A'
THEN
  Ch2 := Ch3
```

is an <IF statement>.

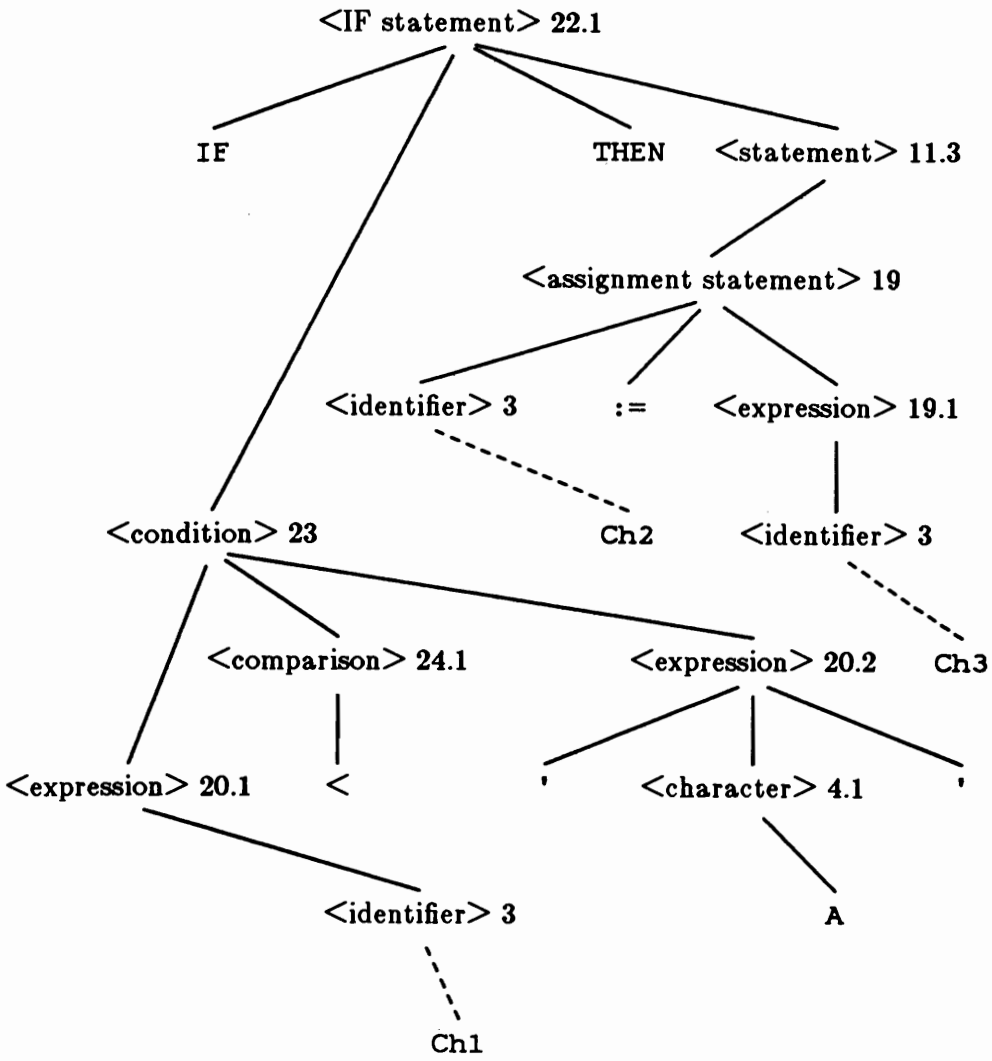


Figure 11: Syntax Tree for an <IF statement>

The reader should check that the rules listed in Figure 11 are correctly used, and that the string can be obtained from the leaves as claimed.

Figure 11 could be drastically abbreviated as shown in Figure 12.

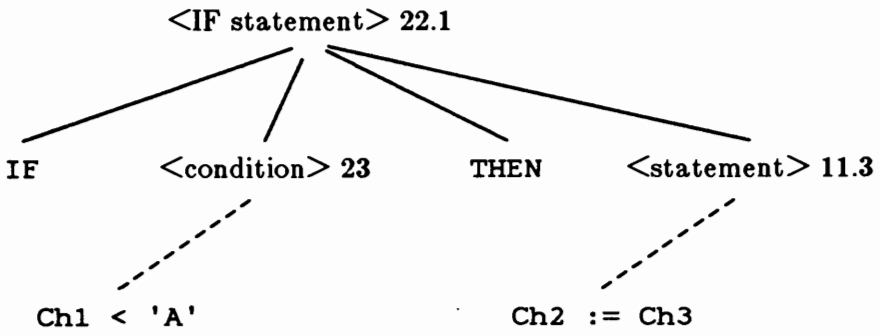


Figure 12: An Abbreviated Version of Figure 11

The choice of which tree will be more helpful in showing that this fragment is syntactically correct depends on the experience of the reader.

Comments in Pascal programs have not been mentioned in the BNF syntax rules, because they have a very special rule:

> A comment begins with the <special character> { and ends with }; between these markers any <character>s may appear. A comment may be placed in an otherwise syntactically correct <program> at any position where a blank could appear, except within a <character string> of a <write item>.

The Pascal machine will attempt to execute any string satisfying the syntax and context rules as a CF Pascal program. Correct syntax is necessary for execution, but it in no way guarantees the results. The syntactically correct program may go into an endless iteration, it may abort because a READ statement calls for more data than is available, it may solve the wrong problem, etc. These problems lie beyond syntax, in another domain called *semantics,* which is the subject of the rest of this book.

2.2.1 Exercises

2.2.1 In each part of this exercise, choose the BNF syntax part <*X*> which most directly can be used to construct the given string of characters. Give a syntax tree to show that indeed the given character string is an <*X*>. Use the following abbreviations: do not write out the details for <identifiers>; where a part has appeared earlier, use it directly without redrawing the tree.

a) `PROGRAM Ups(INPUT,OUTPUT)`

b) `Ch1, Ch2`

c) `VAR Ch1, Ch2: CHAR`

d) `READ(Ch1, Ch2)`

e) `Ch1 < Ch2`

f) `Ch2 <> '#'`

g) `Ch1 := Ch2`

h) `WRITE(Ch1)`

i)
```
IF Ch1 < Ch2
THEN
  WRITE(Ch1)
```

j) `READ(Ch2)`

k)
```
BEGIN
  IF Ch1 < Ch2
  THEN
    WRITE(Ch1)
  Ch1 := Ch2;
  READ(Ch2)
END
```

l)
```
WHILE Ch2 <> '#'
DO
  BEGIN
    IF Ch1 < Ch2
    THEN
      WRITE(Ch1);
    Ch1 := Ch2;
    READ(Ch2);
  END
```

m)
```
PROGRAM Ups(INPUT,OUTPUT);
  VAR
    Ch1, Ch2: CHAR;
  BEGIN
    READ(Ch1, Ch2);
    WHILE Ch2 <> '#'
    DO
      BEGIN
        IF Ch1 < Ch2
        THEN
          WRITE(Ch1);
        Ch1 := Ch2;
        READ(Ch2);
      END
    WRITELN
  END.
```

2.2.2 In each part of this exercise, the given string of characters is almost, but not quite, a syntactically correct part of a Pascal program. Identify a syntax rule that would come close to constructing a syntax tree for each part, and explain what is wrong with the string for that rule.

a)
```
'A' << Ch
```

b)
```
'A' := Ch
```

c)
```
IF 'A' <= Ch
ELSE
  Ch := 'B'
```

d)
```
WHILE 'A' < Ch
  READ(Ch)
```

e)
```
READ(Ch, 'A')
```

f)
```
Ch1; Ch2
```

g)
```
IF Ch1 := Ch2
THEN
   READ (Ch2)
```

h) `'A`

i) `Ch1 := 'AB'`

2.2.3 Determine the syntax correctness of the following examples by attempting to create syntax trees for them. When no tree can be found, give an argument why this is so; otherwise, draw the tree.

a) `BEGIN Ch1 := Ch2; Ch1 := Ch2 END`

b) `BEGIN Ch1 := Ch2, Ch1 := Ch2 END`

c) `IF Ch1 := Ch2 THEN Ch2 := Ch1`

d) `IF 'A' < 'B' THEN Ch1 := Ch1`

e) `WHILE 'A' < 'B' DO BEGIN END`

f) `WHILE 'A' < 'B' DO BEGIN; END`

2.2.4 Write out a *complete* syntax tree for the <program> `CopyChar` whose abbreviated tree is given in this section. Abbreviate for <identifier>s.

2.2.5 Write out an abbreviated syntax tree for the <program> `CopyOdds` given in Section 1.5.3 (Program 24). Use the abbreviations to emphasize the <statement> structure of the program at the expense of other details.

2.2.6 Not only are there recursive definitions in the syntax rules (such as for <identifier>, <identifier list>, etc.), but there is additional recursion among the rules. For example, <BEGIN statement> is defined using <statement>, which is defined using <BEGIN statement>.

Determine which syntax classes are defined recursively in this broader sense. That is, which syntax classes have definitions which, when pursued step by step, can possibly lead back to the syntax class itself?

2.2.7 Develop an abbreviated syntax tree for the <program> `MaxOfAll` (Section 1.5.3, Program 23) using your best judgment in deciding what to abbreviate. Then develop separate trees for each abbreviation (possibly abbreviating these as well), and so on, until you have developed a complete tree for `MaxOfAll`.

2.2.8 Give a rule of thumb for estimating the number of lines that are needed to draw a *complete* syntax tree, for a <program>, in terms of the number of lines in the <program>, its length of identifiers, and so on.

2.3 Program Design and Development

Preview

Knowing the correct syntax of Pascal statements is a beginning, but to write a correct program requires a careful plan for what the program will do when it executes. Plans for a program can be written informally as designs and then the details of the designs can be supplied in a stepwise refinement process. An orderly design process accumulates design parts into executable programs of increasing size. Designs can be tested during development through the execution of well-considered test cases.

New ideas: program outlines, stepwise refinement, design part, program development, program testing.

Programming is a form of writing, and the principles of good writing apply to programming as well. However, programs are read and executed mechanically and relentlessly by a Pascal machine, so the writing must state precisely what is intended. This requirement for precision does not weaken the need for following the rules of good writing, but it does strengthen the value of one rule especially—the rule of writing with designs and sticking to the subject of the design at hand.

We intentionally use the expression "writing *with* designs" rather than "writing *from* designs" because it is not always possible to create a complete design first and then follow it to the letter.

But when a design proves inadequate, it is essential to change it and continue following the updated version. Otherwise programming wanders off into details, and all control is lost.

Designs have a hierarchical structure of increasing detail. This structure allows the writer to zoom down into the details of one part of the design and then to zoom out again without getting lost in the details. It allows the writer to continuously examine and reflect on the overall composition while the writing is going on. If the writing is not going well, perhaps the design should be changed to respond to an unexpected difficulty.

In natural-language writing the designing or outlining process goes only so far before the composition of paragraphs and sentences begins. Programming is more precise, and designs will carry through right down to the final program text. The whole of programming can be regarded as a design process. We call that designing process *stepwise refinement.* The term stepwise refinement was coined by Niklaus Wirth, the designer of Pascal. Each step in the composition process describes in a new level of detail what the program is to be. The final steps describe every detail of the CF Pascal text.

Good design organizes the writing of program text so that intellectual control of the task is as easy as possible. By designing a program one step at a time, the chance of forgetting something is reduced, and it is unnecessary to remember so many details.

Designs in the stepwise refinement of a program are written in a special and precise way. CF Pascal text with correct syntax always forms the backbone of the design. Where low-level details are not yet appropriate, a comment describes the task to be accomplished. Then at the next level this comment is replaced by other CF Pascal text, perhaps with other comments hiding further details. Those are then expanded, and so on. The final program will then consist of the CF Pascal text from each step, and the original comments are retained to describe what the new text inserted at each step does. In the end, every comment has its implementing Pascal text, and the program is ready to run.

2.3.1 Paul Revere's Ride

Listen, my children, and you shall hear
Of the midnight ride of Paul Revere,
On the 18th of April, in seventy-five;
Hardly a man is now alive
Who remembers that famous day and year.

He said to his friend, "If the British march
By land or sea from the town tonight,
Hang a lantern aloft in the belfry arch
Of the North Church tower as a signal light,—

One, if by land, and two, if by sea;
And I on the opposite shore shall be,
Ready to ride and spread the alarm
Through every Middlesex village and farm,
For the country folk to be up and to arm."

- Henry Wadsworth Longfellow

In Paul Revere's honor, a program will be designed to read a single character (assumed to be a **1** or a **2**) from the input. If the character is **1**, the program is to issue this message:

```
The British are coming by land.
```

On input **2**, the message is to be

```
The British are coming by sea.
```

Every <program> must have a name in the program <header>, and here `PaulRevere` is appropriate:

```
PROGRAM PaulRevere(INPUT,OUTPUT)
```

The plan is to read the input into a variable, which must be declared in the program's <block> Lanterns seems a good name, since those were the lights Paul Revere was looking for. Then t program will have to examine the input value and produce the proper message, but the details that will be deferred. Thus we have the first design in the scheme of stepwise refinement:

```
PROGRAM PaulRevere(INPUT,OUTPUT);
  {Print appropriate messages depending on the value
   in INPUT:  'By land' for 1; 'By sea' for 2.}
  VAR
    Lanterns: CHAR;
  BEGIN {PaulRevere}
    {Read Lanterns};
    {Issue Paul Revere's message}
  END.  {PaulRevere}
```

All that remains is to refine the two comments in the <BEGIN statement>. Is the syntax correct s far? Just to check, Figure 13 gives a syntax tree.

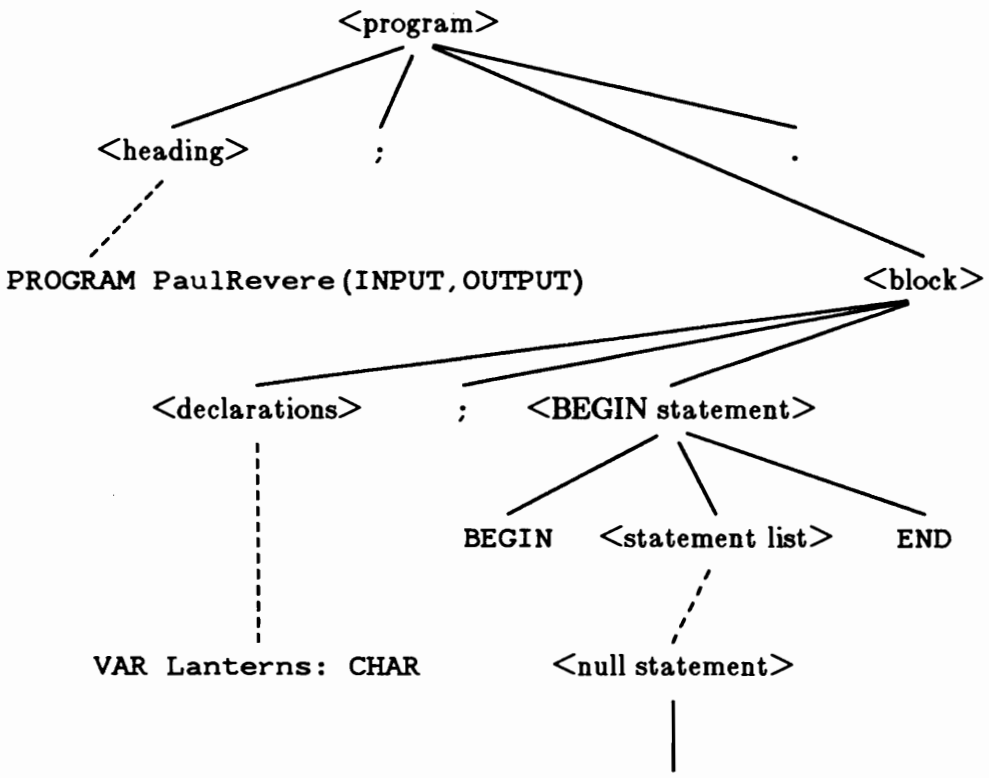


Figure 13: Syntax Tree for PaulRevere

Reading the character strings from this tree, inserting blanks, comments, and line breaks according to the rules, gives back the original program design. Since PaulRevere is syntactically correct so far, the program could be executed. The <null statement> reads no input, and produces no output, so the execution doesn't solve the problem for which the program was intended, but giving it to the Pascal machine would check for typographical errors in transferring program to computer.

A slight addition to the program can provide some useful output, reading and echoing the input:

```
PROGRAM PaulRevere(INPUT,OUTPUT);
  {Print appropriate messages depending on the value
   in INPUT:  'By land' for 1; 'By sea' for 2.}
  VAR
    Lanterns: CHAR;
  BEGIN {PaulRevere}
    {Read Lanterns}
    READ(Lanterns);
    WRITELN('Lanterns is ', Lanterns); {Test statement}
    {Issue Paul Revere's message}
  END.  {PaulRevere}
```

Executions

```
INPUT :1
OUTPUT:Lanterns is 1

INPUT :X
OUTPUT:Lanterns is X
```

Now it is time to take the next step in the stepwise refinement, to take the action called for by the comment after WRITELN in PaulRevere. After Lanterns has been set by the READ statement, its value can be checked with an IF statement:

```
{Issue Paul Revere's message}
IF Lanterns = '1'
THEN
  WRITELN('The British are coming by land.')
ELSE
  WRITELN('The British are coming by sea.')
```

This IF statement assumes that if Lanterns does not have value 1, then it must have value 2. It would be better to be more careful about this:

```
{Issue Paul Revere's message}
IF Lanterns = '1'
THEN
  WRITELN('The British are coming by land.')
ELSE
  IF Lanterns = '2'
  THEN
    WRITELN('The British are coming by sea.')
  ELSE
    WRITELN('The North Church shows only ''',Lanterns,'''.')
```

Now the program checks its input and doesn't send Paul riding off on a wild goose chase. The lesson is that if incorrect data are to be handled correctly, they must be expected and programmed for as carefully as for correct data. Putting this compound IF statement in place of the test statement that only echos the input suggests an improved comment:

```
PROGRAM PaulRevere(INPUT,OUTPUT);
  {Print appropriate messages depending on the value
   in INPUT:  'By land' for 1; 'By sea' for 2;
   otherwise an error message.}
  VAR
    Lanterns: CHAR;
  BEGIN {PaulRevere}
    {Read Lanterns}
    READ(Lanterns);
    {Issue Paul Revere's message}
    IF Lanterns = '1'
    THEN
      WRITELN('The British are coming by land.')
    ELSE
      IF Lanterns = '2'
      THEN
        WRITELN('The British are coming by sea.')
      ELSE
        WRITELN('The North Church shows only ''',Lanterns,'''.')
  END. {PaulRevere}
```

```
INPUT :1
OUTPUT:The British are coming by land.

INPUT :A
OUTPUT:The North Church shows only 'A'.

INPUT :2
OUTPUT:The British are coming by sea.
```

All three executions are required to try all the cases in the program's IF statement.

2.3.2 Design Parts and Development Programs

In a small program like PaulRevere, stepwise refinement can be done in an informal way. However, as problems and programs grow more complex, an orderly and systematic approach saves time in the end. For very complex problems, an orderly and systematic approach means the difference between solving the problem with a program or failing to solve it.

Programming has two distinct phases:

1) *Design* —thinking out what the program should be in order to solve the problem.

2) *Development* —putting the program text in exact form for use by the Pascal machine.

In the stepwise refinement of a program, the CF Pascal text designed to carry out a task in more detail will be called a *design part*. A design part may itself contain more detailed task descriptions (in the form of comments) to be carried out by additional design parts. The result of the design phase will be a hierarchy of design parts which collectively solve the problem at hand. Each design part will be a CF Pascal statement—usually a BEGIN statement, an IF statement, or a WHILE statement—typically of 5 to 15 lines each, perhaps with two to four tasks left to be designed at the next level.

After the program has been entirely refined into a hierarchy of design parts, the translation into machine-readable form can begin. At first glance, this might appear to be an easy step. However, if a large program is translated into Pascal with even one error, the outcome can be a disaster. The best that can happen is that incorrect syntax will be detected by the Pascal compiler. That gives the programmer a second chance to translate. If the program goes into execution, it may produce obviously incorrect results, but the programmer may not suspect the translation and can waste time examining the design for flaws. The results can be so bizarre that it is very difficult to

trace them to the source. For example, use of a wrong <condition>, e.g., < instead of >, can send execution into an unexpected part of the program, a part written with something else in mind, with quite undreamed of consequences. Perhaps the worst outcome is that the program *appears* to work properly, because its tests do not expose the flaw in translation. Then the difficulties will arise at some later time, unexpectedly, and when the design is not fresh in anyone's mind.

A sequence of executable programs, each reflecting a larger part of the design, can facilitate orderly and systematic translation into Pascal. Each program in such a sequence will be called a *development program*. Development programs are accumulations of design parts, which grow in size until the entire design has been turned into Pascal. Each development program is defined so that it can be executed and tested to verify correct translation at each step of development.

In practice, the translation of a design into Pascal can be done in chunks larger than one design step, typically 15 to 50 lines at a time. That is, each successive development program is created by combining a few more design parts with the last tested development program. In order to verify correct execution, it may be necessary to include temporary WRITE statements to create visible output.

2.3.3 Sarah Revere's Letter

Longfellow did not mention Paul Revere's cousin, Sarah Revere, because she has just been invented. But suppose that she discovered the British route early enough to mail a letter to her cousin Paul. Since the British could intercept and read letters before passing them on, Sarah sent her message in code: whichever character string (**land** or **sea**) appears first in the letter is the message. For example, if Sarah's letter starts out

"Dear Paul: I have been to the Chelsea headland..."

then the British are coming by sea, since Chel**sea** appears before head**land**.

The program to be designed and developed in this section is to decode Sarah's letter and create the correct message in English as before.

2.3.3.1 The Design Phase for Sarah Revere's Letter

There are many ways to go about solving this problem. One way is to look for the letter **l** or **s** and then to check the next few letters to see if the character string **and** or **ea** follows, respectively. However, in the string Chelsea, we would have to be careful not to miss the **sea** while looking for **and** after the **l**. A simpler way is to pass a four-character "window" along Sarah's letter, moving it one character at a time. Then we can look for the strings **land** and **sea** directly in the window. For example, in the above letter the successive window contents are:

Dear, **ear**□, **ar**□**P**, **r**□**Pa**, □**Pau**, **Paul**,

and so on, where blanks have been shown as □. The window can be held in four character variables (say W1, W2, W3, and W4). To get from one window view to the next, move the three rightmost variables each one step to the left and then read the next input character into the rightmost variable:

```
BEGIN {Move window}
  W1 := W2;
  W2 := W3;
  W3 := W4;
  READ(W4)
END
```

Once four characters are in W1, W2, W3, W4, check for **land** with this IF statement:

```
{Check for 'land'}
IF W1 = 'l'
THEN
  IF W2 = 'a'
  THEN
    IF W3 = 'n'
    THEN
      IF W4 = 'd'
      THEN {'land' has been found}
```

Checking for **sea** is similar, except that the three characters should be in the last three variables of the window.

This preliminary thinking and trial statement writing suggests a general strategy for decoding Sarah's letter. The embryo outline consists of repeating three steps:

```
WHILE {'land' or 'sea' not found}
DO
  BEGIN
    {move window};
    {check window for 'land'};
    {check window for 'sea'}
  END
```

The iteration must be started with a window and stopped when **land** or **sea** is found or the end of the letter is reached. To start, we could read the first four characters of Sarah's letter into W1, W2, W3, W4. But this presumes that there are four characters to read, which is very likely but not logically necessary. The alternative is to set all four characters of the window to blanks and move the window a character at a time. The window move has to contend with the end of data at some point, and that capability can be used from the very start. It will take four iterations to fill the window to begin with, but that will do no harm; and certainly **land** or **sea** will not be found by mistake while filling the window.

The information that the iteration should be stopped will be obtained inside the WHILE statement, but the test itself must be made at its beginning. A variable can solve this dilemma—it can be initialized to the "keep going" state, and set within the WHILE statement to a "stop now" state. The same variable can be used to remember what stopped the iteration. Looking seems an appropriate name for this variable, with values:

Looking	Meaning
Y	keep looking.
L	**land** was found.
S	**sea** was found.
N	at end of input, neither **land** nor **sea** had been found.

With this general strategy in mind, the first design part can be written:

Design Part 1

```
PROGRAM SarahRevere(INPUT,OUTPUT);
  {Print a message about how the British are
   coming, depending on whether 'land' or 'sea'
   appears first in INPUT.}
  VAR
    W1, W2, W3, W4, Looking: CHAR;
  BEGIN {SarahRevere}
    {Initialize W1, W2, W3, W4, Looking};
    WHILE Looking = 'Y'
    DO
      BEGIN
        {Move window, check end of data};
        {Check window for 'land'};
        {Check window for 'sea'}
      END;
    {Create Sarah's message}
  END. {SarahRevere}
```

This design part has correct syntax, with the English statements written as comments. The next step is to replace these comments by CF Pascal BEGIN statements that carry out the intentions expressed in the English statements. The comments become annotations in the BEGIN statements of design parts. First, the window variables and Looking can be initialized as planned.

Design Part 1.1

```
BEGIN {Initialize W1, W2, W3, W4, Looking}
  W1 := ' ';
  W2 := ' ';
  W3 := ' ';
  W4 := ' ';
  Looking := 'Y'
END
```

Next, the window can be moved as planned, but the end of input must be detected as well. Assuming that the end of data is marked by the first #, it will first show up in W4.

Design Part 1.2

```
BEGIN {Move window, check end of data}
  W1 := W2;
  W2 := W3;
  W3 := W4;
  READ(W4);
  IF W4 = '#'
  THEN {End of Data}
    Looking := 'N'
END
```

Checking the window for **land** has also been planned out, and Looking must be set if it is found.

Design Part 1.3

```
BEGIN {Check window for 'land'}
  IF W1 = '1'
  THEN
    IF W2 = 'a'
    THEN
      IF W3 = 'n'
      THEN
        IF W4 = 'd'
        THEN {'land' has been found}
          Looking := 'L'
END
```

sea will first appear in the window at the right, so:

Design Part 1.4

```
BEGIN {Check window for 'sea'}
  IF W2 = 's'
  THEN
    IF W3 = 'e'
    THEN
      IF W4 = 'a'
      THEN {'sea' has been found}
        Looking := 'S'
END
```

The iteration will continue unless `Looking` is set within it, to one of the values **L**, **S**, or **N**. This value will stop the WHILE statement's execution, and allow Sarah's message to be deciphered.

Design Part 1.5

```
BEGIN {Create Sarah's message}
  IF Looking = 'L'
  THEN
    WRITELN('The British are coming by land.')
  ELSE
    IF Looking = 'S'
    THEN
      WRITELN('The British are coming by sea.')
    ELSE
      WRITELN('Sarah didn't say')
END
```

Design Parts 1.1 to 1.5 fill all the gaps in Design Part 1, so the design is complete. None of the second-level designs were complex enough to require a third level of stepwise refinement.

2.3.3.2 The Development Phase for Sarah Revere's Letter

The entire <program> `SarahRevere` could be assembled into one piece of text immediately. But the six design parts are easier to work with and change if any difficulties should be encountered. Eventually, the entire program must be assembled for the Pascal machine, but it is wise to do it carefully. The design parts can be used to create a sequence of programs, each of which can be tested to check that the design works, and its translation into Pascal has been correctly accomplished. The first three design parts together move the four-character window along the input. The proper operation of this window motion can be checked by executing Program 1A, which includes a WRITE statement printing each window as it is created. The successive window values are separated by commas to make it easier to read the output. A `WRITELN` is also needed outside the WHILE statement, so that the new output will appear.

Development Program 1A

```
PROGRAM SarahRevere(INPUT,OUTPUT);
  {Print a message about how the British are
   coming, depending on whether 'land' or 'sea'
   appears first in INPUT.}
  VAR
    W1, W2, W3, W4, Looking: CHAR;
  BEGIN {SarahRevere}
    BEGIN {Initialize W1, W2, W3, W4, Looking}
      W1 := ' ';
      W2 := ' ';
      W3 := ' ';
      W4 := ' ';
      Looking := 'Y'
    END;
    WHILE Looking = 'Y'
    DO
      BEGIN
        BEGIN {Move window, check end of data}
          W1 := W2;
          W2 := W3;
          W3 := W4;
          READ(W4);
          IF W4 = '#'
          THEN {End of data}
            Looking := 'N';
        END;
        WRITE(W1, W2, W3, W4, ','); {Test statement}
        {Check window for 'land'};
        {Check window for 'sea'}
      END;
    WRITELN; {Test statement}
    {Create Sarah's message}
  END. {SarahRevere}
```

Executions

```
INPUT :#
OUTPUT:    #,

INPUT :D#
OUTPUT:    D,   D#,

INPUT :De#
OUTPUT:    D,   De,  De#,

INPUT :Dea#
OUTPUT:    D,   De,  Dea,Dea#,

INPUT :Dear Pau#
OUTPUT:    D,   De,  Dea,Dear,ear ,ar P,r Pa,  Pau,Pau#
```

Development Program 1A is 33 lines long—well within our rule-of-thumb limit of 50 lines. Six of those lines are comments, and there are two repetitive patterns in the initialization and window-moving parts. The comment {Test statement} serves as a reminder that the WRITE statements will be removed later.

Development Program 1A executed as we expected it to. If it had not, the first step would have been to check the mechanical process of combining the design parts to form the final text. If that seemed all right, it would be time to reexamine the designs. By combining only some design

parts, the process of detecting and correcting errors is simplified. In any case, before more design parts are incorporated, the current development program must do what is expected of it.

Design Parts 2.3 and 2.4 are incorporated in Development Program 1B, in which the window-test WRITE statements of Program 1A have been replaced by a new test WRITE statement for the value of Looking. Development Program 1B adds 18 lines to Development Program 2A. The next larger alternative would be to include Sarah's message as well—10 more lines. We have chosen the smaller step, but the design parts are quite simple, and the larger alternative would be reasonable. The following test executions exercise all three options for Looking.

Development Program 1B

```
PROGRAM SarahRevere(INPUT,OUTPUT);
  {Print a message about how the British are
   coming, depending on whether 'land' or 'sea'
   appears first in INPUT.}
  VAR W1, W2, W3, W4, Looking: CHAR;
  BEGIN {SarahRevere}
    BEGIN {Initialize W1, W2, W3, W4, Looking}
      W1 := ' ';
      W2 := ' ';
      W3 := ' ';
      W4 := ' ';
      Looking := 'Y'
    END;
    WHILE Looking = 'Y'
    DO
      BEGIN
        BEGIN {Move window, check end of data}
          W1 := W2;
          W2 := W3;
          W3 := W4;
          READ(W4);
          IF W4 = '#'
          THEN {End of data}
            Looking := 'N'
        END;
        BEGIN {Check window for 'land'}
          IF W1 = 'l'
          THEN
            IF W2 = 'a'
            THEN
              IF W3 = 'n'
              THEN
                IF W4 = 'd'
                THEN {'land' has been found}
                  Looking := 'L'
        END;
```

```
        BEGIN {Check window for 'sea'}
          IF W2 = 's'
          THEN
            IF W3 = 'e'
            THEN
              IF W4 = 'a'
              THEN {'sea' has been found}
                Looking := 'S'
        END
      END;
    WRITELN('Looking is ', Looking); {Test statement}
    {Create Sarah's message}
  END. {SarahRevere}
```

Executions

```
INPUT :Dear Paul: I have been #
OUTPUT:Looking is N

INPUT :Dear Paul: I have been to Chelsea #
OUTPUT:Looking is S

INPUT :Dear Paul: My land lies along the sea #
OUTPUT:Looking is L
```

Development Program 1B appears to execute correctly, so the development can be completed by adding Design Part 2.5 and removing the test WRITE statement. The result in Development Program 1C should be a correctly executing final program:

(Final) Development Program 1C

```
PROGRAM SarahRevere(INPUT,OUTPUT);
  {Print a message about how the British are
   coming, depending on whether 'land' or 'sea'
   appears first in INPUT.}
  VAR W1, W2, W3, W4, Looking: CHAR;
  BEGIN {SarahRevere}
    BEGIN {Initialize W1, W2, W3, W4, Looking}
      W1 := ' ';
      W2 := ' ';
      W3 := ' ';
      W4 := ' ';
      Looking := 'Y'
    END;
    WHILE Looking = 'Y'
    DO
      BEGIN
        BEGIN {Move window, check end of data}
          W1 := W2;
          W2 := W3;
          W3 := W4;
          READ(W4);
          IF W4 = '#'
          THEN {End of data}
            Looking := 'N'
        END;
```

```
        BEGIN {Check window for 'land'}
          IF W1 = 'l'
          THEN
            IF W2 = 'a'
            THEN
              IF W3 = 'n'
              THEN
                IF W4 = 'd'
                THEN {'land' has been found}
                  Looking := 'L'
        END;
        BEGIN {Check window for 'sea'}
          IF W2 = 's'
          THEN
            IF W3 = 'e'
            THEN
              IF W4 = 'a'
              THEN {'sea' has been found}
                Looking := 'S'
        END
      END;
    BEGIN {Create Sarah's message}
      IF Looking = 'L'
      THEN
        WRITELN('The British are coming by land.')
      ELSE
        IF Looking = 'S'
        THEN
          WRITELN('The British are coming by sea.')
        ELSE
          WRITELN('Sarah didn''t say')
    END
  END. {SarahRevere}
```

Executions

```
INPUT :Dear Paul: I have been #
OUTPUT:Sarah didn't say

INPUT :Dear Paul: I have been to Chelsea #
OUTPUT:The British are coming by sea.

INPUT :Dear Paul: My land lies along the sea #
OUTPUT:The British are coming by land.
```

Creating design parts and a development programs is a process of analysis and synthesis. In the analysis (design) phase, a general problem is decomposed into smaller problems at each step until only CF Pascal text remains. In the synthesis (development) phase, larger pieces of text are assembled at each step until the final program is created. The assembly of text parts proceeds in a special way so that the construction can be tested at each step.

2.3.4 Principles of Program Design and Development

This section summarizes the principles of design by stepwise refinement, and development with testing.

DESIGN PHASE

Problem Solving (in parallel with program design)

1. *Understand the problem.* If necessary, write a concise problem statement and keep it updated continuously as you discover how to solve it.
2. *Develop a solution strategy.* Outline a solution in English, in one or more levels of detail. Keep the solution strategy updated as you create your final program design.
3. *Analyze and criticize your solution strategy.* At every level of detail, look for simpler ways to solve the problem. Is your solution no more complex than the problem demands?
4. *Organize your solution* by writing improved designs. Should some design part be reorganized for better understandability? Can similar operations be found and organized into common design parts?

Program Design (in parallel with problem solving)

1. *Develop a writing plan* by converting your solution strategy into a hierarchy of design parts made up of program text and brief task descriptions. These task descriptions should serve in turn as names for design parts at the next level of detail.
2. As a rule, *keep design parts to 15 lines or less,* with coherent and easily understood task descriptions. Keeping design parts small and coherent takes discipline, but it is well worth the effort.
3. *Keep your design parts physically separate* for writing and rewriting flexibility, but think of them in a hierarchical structure when analyzing and simplifying your program.
4. *Write and rewrite your design parts and your guiding solution strategy.* Solve your problem, put the solution away, come back to it and improve it.

DEVELOPMENT PHASE

Program Development (in parallel with program testing)

1. *Develop a construction plan.* Assemble design parts (beginning with a simple program) into a sequence of larger and larger programs. The plan should be suitable for execution a step at a time.
2. As a rule, *add no more than 50 lines of design to a development at one time.*
3. *Don't start on the first 50 lines until you have your design completed.* It should be the last design you have written out—not the first.
4. *Get every 50 lines right the first time.* Concentration now will save time and frustration later.

Program Testing (in parallel with program development)

1. *Think as hard about your construction plan as about your final program.* Create the right sequence of programs to give positive feedback at each step of the construction.
2. *Develop a test plan.* Create programs with temporary WRITE statements to get meaningful execution results (although not necessarily the final results that you are looking for).
3. *Create test cases for every step of your construction plan,* and don't forget the exceptional cases—even though they may seem unlikely.

2.3.5 Exercises

2.3.1 In each part modify the <program> `PaulRevere` as required.

a) To use a single WRITE statement for printing "The British are coming by" part of the message.

b) To use inputs `L` and `LL` (for one or two lanterns) instead of `1` and `2`.

c) To include an input of `3` to stand for "The British are coming by air" (via balloon, of course!).

2.3.2 In the course of developing `PaulRevere`, a test WRITE statement was inserted to echo the input. Explain why it would be superfluous to retain this statement as part of the final program.

2.3.3 For each part, break the named program into appropriate design parts and development programs.

a) `MaxMin` (Program 20, Section 1.4.3).

b) `MaxOfAll` (Program 23, Section 1.5.3).

c) `DoubleChars` (Program 25, Section 1.6.1).

2.3.4 When Sarah Revere was thinking about how to send her coded message, why didn't she make things much simpler by using the first occurrence of the single letter **`1`** or **`s`** as the code?

2.3.5 Write a design part, which could be added to `SarahRevere`, to check for the possibility of the British coming by air.

2.3.6 Each part below suggests a change in `SarahRevere`. For each answer: (i) Will the changed program give the same results as the original? (ii) What is the crucial test case to examine? (iii) Is the change one of design or development?

a) Place the statements to move the window at the bottom of the DO statement.

b) Check for **`Sea`** using `W1, W2, W3` instead of `W2, W3, W4`.

c) Both (a) and (b).

2.3.7 Suppose that in solving Sarah Revere's problem, it had been decided to move a three-character window, and should **`lan`** appear in it, to make a special check for **`d`** following. Where does such a change fit in the process of design and development? Make the change in the appropriate place. Is the resulting program better?

2.3.8 As `SarahRevere` stands, it can be fooled by changes of case; for example, it will report "Sarah didn't say" for the input **`Land Ho!`**. Modify the program to allow all changes of case in the code words, and discuss whether this is a design or development change.

2.3.9 In the test executions of Development Program 1B, why isn't the value **`Y`** for `Looking` tested along with the others?

2.4 Program Organization with Procedures

Preview

As programs grow larger, design part organization becomes a major technique in creating a comprehensible program under intellectual control. CF Pascal offers convenient procedure declarations to describe design parts and procedure statements to execute the declared procedures.

New ideas: procedure declaration, procedure statement, appearance rule, copy rule, reusing procedures.

2.4.1 Simple Procedures

CF Pascal provides a useful mechanism for grouping and executing <statement>s, the *procedure.* The organization of a problem solution can make use of procedures to name each design part, and then to use the name in development. A BEGIN statement can be named in a *procedure declaration,* using a standard word PROCEDURE. For example,

```
PROCEDURE Copy;
  BEGIN {Copy}
    READ(Ch);
    WRITE(Ch)
  END   {Copy}
```

is a Pascal procedure declaration. This procedure's name is the <identifier> Copy. Procedure declarations must be placed in a program just after the data declarations; any number may be given, separated by semicolons.

Once a procedure declaration has been given in a program, a *procedure statement* can be placed in the *program's* BEGIN statement (not the BEGIN statment in the procedure declaration). The procedure statement has the form of the procedure name standing alone. The effect of executing the procedure statement is the effect of the BEGIN statement named in its procedure declaration.

For example,

```
PROGRAM UsingCopy(INPUT,OUTPUT);
  {Transfer two characters from INPUT to OUTPUT.
   The program aborts if INPUT is empty.}
  VAR
    Ch: CHAR;
  PROCEDURE Copy;
    BEGIN {Copy}
      READ(Ch);
      WRITE(Ch)
    END;  {Copy}
  BEGIN {UsingCopy statements begin here}
    Copy;
    Copy;
    WRITELN
  END. {UsingCopy}
```

Execution

```
INPUT :ABCDE
OUTPUT:AB
```

UsingCopy will transfer two characters, just as if the BEGIN statement of Copy appeared twice in the BEGIN statement of UsingCopy. The statements of a procedure are not executed when declared; they are executed when the procedure is named in a procedure statement during the execution of the program's BEGIN statement.

The execution table for a program with procedures reflects this delayed execution; for example:

Execution Table for `UsingCopy`

After Execution of	INPUT	OUTPUT	Ch
	ABCDE		
`PROGRAM UsingCopy...`	**ABCDE**	_	
`VAR`			
`Ch:CHAR`			?
`BEGIN {UsingCopy...}`			
`Copy`			
`BEGIN`			
`READ(Ch)`	**ABCDE**		**A**
`WRITE(Ch)`		**A_**	
`END`			
`Copy`			
`BEGIN`			
`READ(Ch)`	**ABCDE**		**B**
`WRITE(Ch)`		**AB_**	
`END`			
`WRITELN`		**AB/_**	
`END.`	**ABCDE**	**AB**	

As another example, `PaulRevere` could have been designed with procedures:

```
PROGRAM PaulRevere2(INPUT,OUTPUT);
  VAR
    Lanterns: CHAR;
  PROCEDURE IssueMessage;
    BEGIN {IssueMessage}
      IF Lanterns = '1'
      THEN
        WRITELN('The British are coming by land.')
      ELSE
        IF Lanterns = '2'
        THEN
          WRITELN('The British are coming by sea.')
        ELSE
          WRITELN('The North Church shows only ''', Lanterns,'''.')
    END; {IssueMessage}
  BEGIN {PaulRevere2}
    {Read Lanterns}
    READ(Lanterns);
    {Issue Paul Revere's message}
    IssueMessage
  END.  {PaulRevere2}
```

Procedures can be added to the formal syntax of CF Pascal by changing Syntax Rules 6 and 11 and adding three new rules:

SR6. <block> ::= <BEGIN statement>
¦ <declarations> ; <BEGIN statement>
¦ <procedure list> ; <BEGIN statement>
¦ <declarations> ; <procedure list> ; <BEGIN statement>

SR11. <statement> ::= <READ statement>
| <WRITE statement>
| <assignment statement>
| <null statement>
| <BEGIN statement>
| <IF statement>
| <WHILE statement>
| <procedure statement>

SR26. <procedure list> ::= <procedure>
| <procedure list> ; <procedure>

SR27. <procedure> ::= PROCEDURE <identifier> ; <BEGIN statement>

SR28. <procedure statement> ::= <identifier>

An abbreviated syntax tree for the <program> `UsingCopy` is shown in Figure 14.

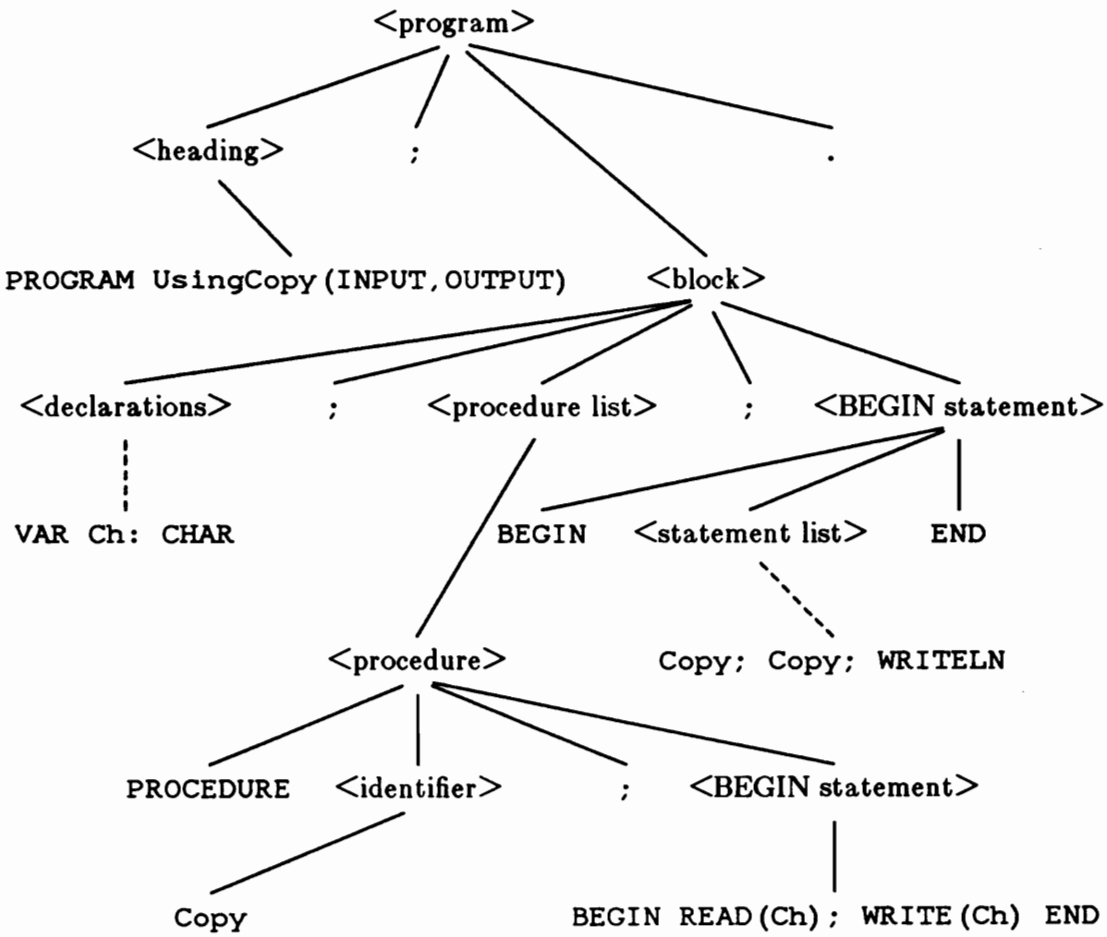


Figure 14: Syntax Tree for `UsingCopy`

Context Rule 3 also requires modification, and some new rules are needed:

CR3. Any <identifier> in a <statement> that is not a <procedure statement>, other than standard identifiers (e.g., READ, WRITE, etc.) must appear in the <identifier list> of the <declarations>.

CR4. (Appearance Rule) The <identifier> of a <procedure statement> must appear as the <identifier> following PROCEDURE in a <procedure> appearing before the <procedure statement>.

CR5. Variable and procedure names can't be the same <identifier>.

Context Rule 4 is called the *appearance rule* because it regulates where procedure declarations must appear (before the corresponding procedure statement). For example, if UsingCopy were given two more procedures:

```
PROGRAM UsingCopyToo(INPUT,OUTPUT);
  VAR
    Ch: CHAR;
  PROCEDURE ReadOnly;
    BEGIN
      READ(Ch)
    END; {ReadOnly}
  PROCEDURE WriteOnly;
    BEGIN
      WRITE(Ch)
    END; {WriteOnly}
  PROCEDURE Copy;
    BEGIN
      ReadOnly;
      WriteOnly
    END; {Copy}
  BEGIN {UsingCopyToo}
    Copy;
    Copy;
    WRITELN
  END. {UsingCopyToo}
```

then <procedure>s ReadOnly and WriteOnly must precede Copy by the appearance rule, since the <procedure statement>s ReadOnly and WriteOnly appear in Copy.

When procedure statements appear within procedures, the execution effect is as before; it is as if the BEGIN statement of the procedure named is copied in place of the procedure statement and executed. This effect is called the *copy rule* for procedure statements.

2.4.2 CountChars Again

CountChars (Program 27, Section 1.6.3) is a good program to reorganize using procedures. The large blocks of nested IF statements needed to increment the three counters are good candidates for procedures:

```
IF Ones = '0' THEN Ones := '1' ELSE
...
IF Ones = '8' THEN Ones := '9' ELSE
IF Ones = '9'
THEN
  BEGIN
    Ones :='0';
    IF Tens = '0' THEN Tens := '1' ELSE
    ...
    IF Tens = '8' THEN Tens := '9' ELSE
    IF Tens = '9'
    THEN
      BEGIN
        Tens := '0';
        IF Hundreds = '0' THEN Hundreds := '1' ELSE
        ...
        IF Hundreds = '8' THEN Hundreds := '9' ELSE
        IF Hundreds = '9' THEN Overflow := 'Y'
      END
  END
```

The portion that adjusts the Hundreds digit can be made into a procedure (say) BumpHundreds; the Tens adjustment into BumpTens (which will invoke BumpHundreds at the end), and similarly BumpOnes will at the end invoke BumpTens. Then counting can be done by the procedure statement: BumpOnes. To satisfy the appearance rule CR4, the procedure declarations must appear in the order BumpHundreds, BumpTens, BumpOnes. Here is the sequence of design parts that might have been devised:

Design Part 2

```
PROGRAM CountChars2(INPUT,OUTPUT);
  VAR
    Ch, Overflow, Ones, Tens, Hundreds: CHAR;
  BEGIN {CountChars2}
    {Initialize Overflow and counters};
    {Count characters};
    {Write results}
  END. {CountChars2}
```

The most important part of the design is the character counting, so we tackle it next:

Design Part 2.1

```
BEGIN {Count characters}
  READ(Ch);
  WHILE Ch <> '#'
  DO
    BEGIN
      BumpCounter;
      READ(Ch)
    END
END
```

Design Part 2.1.1

```
PROCEDURE BumpCounter;
  BEGIN {BumpCounter}
    BumpOnes
  END {BumpCounter}
```

Design Part 2.1.1.1

```
PROCEDURE BumpOnes;
  BEGIN {BumpOnes}
    IF Ones = '0' THEN Ones := '1' ELSE
    IF Ones = '1' THEN Ones := '2' ELSE
    IF Ones = '2' THEN Ones := '3' ELSE
    IF Ones = '3' THEN Ones := '4' ELSE
    IF Ones = '4' THEN Ones := '5' ELSE
    IF Ones = '5' THEN Ones := '6' ELSE
    IF Ones = '6' THEN Ones := '7' ELSE
    IF Ones = '7' THEN Ones := '8' ELSE
    IF Ones = '8' THEN Ones := '9' ELSE
    IF Ones = '9'
    THEN
      BumpTens
  END; {BumpOnes}
```

Design Part 2.1.1.1.1

```
PROCEDURE BumpTens;
  BEGIN {BumpTens}
    Ones := '0';
    IF Tens = '0' THEN Tens := '1' ELSE
    IF Tens = '1' THEN Tens := '2' ELSE
    IF Tens = '2' THEN Tens := '3' ELSE
    IF Tens = '3' THEN Tens := '4' ELSE
    IF Tens = '4' THEN Tens := '5' ELSE
    IF Tens = '5' THEN Tens := '6' ELSE
    IF Tens = '6' THEN Tens := '7' ELSE
    IF Tens = '7' THEN Tens := '8' ELSE
    IF Tens = '8' THEN Tens := '9' ELSE
    IF Tens = '9'
      THEN
        BumpHundreds
  END; {BumpTens}
```

Design Part 2.1.1.1.1.1

```
PROCEDURE BumpHundreds;
  BEGIN {BumpHundreds}
    Tens := '0';
    IF Hundreds = '0' THEN Hundreds := '1' ELSE
    IF Hundreds = '1' THEN Hundreds := '2' ELSE
    IF Hundreds = '2' THEN Hundreds := '3' ELSE
    IF Hundreds = '3' THEN Hundreds := '4' ELSE
    IF Hundreds = '4' THEN Hundreds := '5' ELSE
    IF Hundreds = '5' THEN Hundreds := '6' ELSE
    IF Hundreds = '6' THEN Hundreds := '7' ELSE
    IF Hundreds = '7' THEN Hundreds := '8' ELSE
    IF Hundreds = '8' THEN Hundreds := '9' ELSE
    IF Hundreds = '9'
    THEN
      Overflow = 'Y'
  END {BumpHundreds}
```

The initialization and results designs are simpler:

Design Part 2.2

```
BEGIN {Initialize Overflow and counters}
  Overflow := 'N';
  Ones := '0';
  Tens := '0';
  Hundreds := '0'
END
```

Design Part 2.3

```
BEGIN {Write results}
  IF Overflow = 'N'
  THEN
    WRITELN('Number of characters is ',
      Hundreds, Tens, Ones)
  ELSE
    WRITELN('Number of characters exceeds 999')
END
```

These design parts total about 85 lines—too many to combine into a program all at once. A more reasonable two-step program development uses a test statement in BumpCounter and inhibits the remaining procedure executions by putting the procedure statement BumpOnes (found in BumpCounter) in a comment. The result is Development Program 2A, built from Design Parts 2, 2.2, 2.1, 2.1.1, and 2.3 above. The declaration of BumpCounter appears before the BEGIN statement of the program as required by the appearance rule. The comment on this BEGIN statement points to where execution begins. On execution with any input data containing a #, Development Program 2A should print a line for each time BumpCounter is invoked, and a number of characters of 000.

Development Program 2A

```
PROGRAM CountChars2(INPUT,OUTPUT);
  VAR
    Ch, Overflow, Ones, Tens, Hundreds: CHAR;
  PROCEDURE BumpCounter;
    BEGIN {BumpCounter}
      WRITELN('Got to BumpCounter OK') {test statement}
      {BumpOnes - inhibited procedure statement for test}
    END;  {BumpCounter}
  BEGIN {CountChars2}
    BEGIN {Initialize Overflow and counters}
      Overflow := 'N';
      Ones := '0';
      Tens := '0';
      Hundreds := '0'
    END;
    BEGIN {Count characters}
      READ(Ch);
      WHILE Ch <> '#'
      DO
        BEGIN
          BumpCounter;
          READ(Ch)
        END
    END;
```

```
    BEGIN {Write Results}
      IF Overflow = 'N'
      THEN
        WRITE('Number of characters is ',
        Hundreds, Tens, Ones)
      ELSE
        WRITE('Number of characters exceeds 999')
    END
  END. {CountChars2}
```

Execution

```
INPUT :ABC#
OUTPUT:Got to BumpCounter OK
       Got to BumpCounter OK
       Got to BumpCounter OK
       Number of characters is 000
```

With execution as expected, BumpCounter can be restored to its original form and the remaini design parts added. The result (Development Program 2B) should execute as expected.

Development Program 2B

```
PROGRAM CountChars2(INPUT,OUTPUT);
  VAR Ch, Overflow, Ones, Tens, Hundreds: CHAR;
  PROCEDURE BumpHundreds;
    BEGIN {BumpHundreds}
      Tens := '0';
      IF Hundreds = '0' THEN Hundreds := '1' ELSE
      IF Hundreds = '1' THEN Hundreds := '2' ELSE
      IF Hundreds = '2' THEN Hundreds := '3' ELSE
      IF Hundreds = '3' THEN Hundreds := '4' ELSE
      IF Hundreds = '4' THEN Hundreds := '5' ELSE
      IF Hundreds = '5' THEN Hundreds := '6' ELSE
      IF Hundreds = '6' THEN Hundreds := '7' ELSE
      IF Hundreds = '7' THEN Hundreds := '8' ELSE
      IF Hundreds = '8' THEN Hundreds := '9' ELSE
      IF Hundreds = '9'
      THEN
        Overflow := 'Y'
    END; {BumpHundreds}
```

```
PROCEDURE BumpTens;
  BEGIN {BumpTens}
    Ones := '0';
    IF Tens = '0' THEN Tens := '1' ELSE
    IF Tens = '1' THEN Tens := '2' ELSE
    IF Tens = '2' THEN Tens := '3' ELSE
    IF Tens = '3' THEN Tens := '4' ELSE
    IF Tens = '4' THEN Tens := '5' ELSE
    IF Tens = '5' THEN Tens := '6' ELSE
    IF Tens = '6' THEN Tens := '7' ELSE
    IF Tens = '7' THEN Tens := '8' ELSE
    IF Tens = '8' THEN Tens := '9' ELSE
    IF Tens = '9'
    THEN
      BumpHundreds
  END; {BumpTens}
PROCEDURE BumpOnes;
  BEGIN {BumpOnes}
    IF Ones = '0' THEN Ones := '1' ELSE
    IF Ones = '1' THEN Ones := '2' ELSE
    IF Ones = '2' THEN Ones := '3' ELSE
    IF Ones = '3' THEN Ones := '4' ELSE
    IF Ones = '4' THEN Ones := '5' ELSE
    IF Ones = '5' THEN Ones := '6' ELSE
    IF Ones = '6' THEN Ones := '7' ELSE
    IF Ones = '7' THEN Ones := '8' ELSE
    IF Ones = '7' THEN Ones := '8' ELSE
    IF Ones = '9'
    THEN
      BumpTens
  END; {BumpOnes}
PROCEDURE BumpCounter;
  BEGIN {BumpCounter}
    BumpOnes
  END;  {BumpCounter}
BEGIN {CountChars2}
  BEGIN {Initialize Overflow and counters}
    Overflow := 'N';
    Ones := '0';
    Tens := '0';
    Hundreds := '0'
  END;
  BEGIN {Count characters}
    READ(Ch);
    WHILE Ch <> '#'
    DO
      BEGIN
        BumpCounter;
        READ(Ch)
      END
  END;
```

```
    BEGIN {Write Results}
      IF Overflow = 'N'
      THEN
        WRITE('Number of characters is ',
                    Hundreds, Tens, Ones)
      ELSE
        WRITE('Number of characters exceeds 999')
    END
  END. {CountChars2}
```

2.4.3 Reusing Procedures

Design Parts 2.1.1 to 2.1.1.1.1, called by the procedure statement `BumpCounter`, are a package of four procedures (about 50 lines of text) that can be reused to increment a counter defined by the variables `Hundreds`, `Tens`, and `Ones`. In order to use the package, these variables and `Overflow` must be declared in a program, and the procedures must be included according to the appearance rule.

Consider the problem of printing the natural numbers from 1 to 500, ten numbers to a line. After initializing the counter variables, a print-and-bump action can be repeated until the counter reaches 500. The highest-level design part can use the `BumpCounter` group of procedures as follows:

Design Part 3

```
PROGRAM PrintNaturals(INPUT,OUTPUT);
  VAR
    Counting, Ones, Tens, Hundreds, Overflow: CHAR;
  BEGIN {PrintNaturals}
    {Initialize Counting and counter variables};
    WHILE Counting = 'Y'
    DO
    BEGIN
      {Print natural number found in counter};
      {Start new line for each multiple of 10};
      BumpCounter;
      {Check if counter is 501}
    END
  END. {PrintNaturals}
```

Although `Overflow` will never be used, it must be declared because it appears in the procedures we plan to use; if `Overflow` were not declared, the Pascal compiler would flag a violation of Context Rule 3.

Design Part 3.1

```
BEGIN {Initialize Counting and counter variables}
  Counting := 'Y';
  Ones := '1';
  Tens := '0';
  Hundreds := '0';
  Overflow := 'N'
END
```

Design Part 3.2

```
BEGIN {Print natural number found in counter}
  IF Hundreds = '0'
  THEN {Counter less than 100}
    IF Tens = '0'
    THEN {Counter less than 10}
      WRITE(' ',' ', Ones, ' ')
    ELSE {Counter between 10 and 99}
      WRITE(' ', Tens, Ones, ' ')
  ELSE
    WRITE(Hundreds, Tens, Ones, ' ')
END
```

Design Part 3.2 suppresses printing of leading zeros; for example, when the counter variables are **003**, **3** is printed.

Design Part 3.3

```
BEGIN {Start new line for each multiple of 10}
  IF Ones = '0'
  THEN
    WRITELN
END
```

Design Part 3.4

```
BEGIN {Check if counter is 501}
  IF Hundreds = '5'
  THEN
    IF Tens = '0'
    THEN
      IF Ones = '1'
      THEN
        Counting := 'N'
END
```

Design Parts 3 through 3.4 comprise about 40 lines of text, which is not too much to assemble in a single program skeleton along with the four procedures (which are already checked out and in machine-readable form). Thus, we have designed and can assemble a program of 90 lines (40 new, 50 reused) to print the natural numbers from 1 to 500.

2.4.4 Exercises

2.4.1 Expand the syntax tree for `UsingCopy` given in Section 2.4.1 to a complete derivation (except for the usual <identifier> abbreviations).

2.4.2 Demonstrate the syntactic correctness of the following programs from Section 2.4.1 by giving a syntax tree annotated with the syntax-rule numbers used to create it. Abbreviate the <identifier> derivations.

a) `PaulRevere2`

b) `UsingCopyToo`

2.4.3 Generate the execution table for `UsingCopyToo` (Section 2.4.1) with input **ABC#**.

2.4.4 Restructure the design of `SarahRevere` (Section 2.3) using procedures for the design parts:

```
{Move window, check end of data}
{Check window for land}
{Check window for sea}
```

2.4.5 In Design Part 2.1.1 `BumpCounter` consists of a single procedure statement, calling `BumpOnes`. Explain why it isn't a good idea to simply replace the former with the latter in Design Part 2.1, and eliminate 2.1.1 (Hint: which name should be dropped if this were done?)

2.4.6 Identify a useful design part of the Sarah Revere problem (Section 2.3), and develop it as a procedure.

2.4.7 If the `SarahRevere` program of Section 2.3 fails to work correctly, it will be helpful to examine what it is doing in detail. Insert a use of the counter subroutines developed in Section 2.4.3, that prints the position at which the code word began, starting with the beginning as position 1. What is a reasonable number to print if the result is that Sarah didn't say? In changing the program clearly mark the statements that will be removed later, when the program is made to work. Comment on whether or not the counter subroutines themselves should be removed at that time.

2.4.8 Rewrite, using procedures of your choice, from Chapter 1:

a) `MaxOfAll` (Program 23, Section 1.5.3)

b) `CopyOdds` (Program 24, Section 1.5.3)

c) `DoubleChars` (Program 25, Section 1.6.1)

d) `TwoCharacterWords` (Program 26, Section 1.6.2)

2.4.9 Design and develop a program like `PrintNaturals` that instead prints only the *even* natural numbers {0, 2, 4, ...}. Use as much of Design Part 3 as you can.

2.4.10 Design and develop a program like `PrintNaturals` that reads two counter values, and prints the natural numbers between those limits.

2.4.11 Design and develop a reusable program part that reads one digit from the input and uses the number it represents to control a loop so that its body is performed just that many times.

2.4.12 Use the program part developed in the previous exercise to modify `PrintNaturals` so that it reads one-digit number N, then prints every *Nth* natural number: {0, N, $2N$, $3N$, ...}.

2.5 Chapter Summary

A Pascal program has two quite distinct aspects, its form (syntax) and its meaning (semantics). In solving a problem with a program, perfect syntax is essential. The Pascal compiler will reject any program that is ill-formed by even a single character. When a mistake is made in syntax, the greater danger is that the resulting program still happens to be syntactically correct. That is, it is the wrong program, but the Pascal compiler will not detect the trouble. Fortunately, the rules of syntax are as precise as the penalties for violating them are severe, so with care programs can be checked for the proper form. Program form is straightforward compared to program meaning. For a given problem, which program captures a solution? It is not enough that the program be well formed; it must have the right meaning, so that when executed it solves the problem. Program design is an art requiring human intelligence and training, but there are principles to guide the process.

Most of the syntax of Pascal is described by formal syntax rules, which tell how to construct a syntax tree. Starting with <program> and its rule, the tree is constructed step by step until the character string hanging from its leaves is the program whose syntax is to be checked. However, there are restrictions on the use of Pascal standard words, and on the insertion of blanks into the program string. More important, there are context restrictions that connect one part of the tree to another. For example, an <identifier> used in a <procedure statement> must have appeared in the <procedure list> of the <block> in which it is used. The details of CF Pascal syntax are summarized in Appendix I.

Program design can be organized and controlled through the process of stepwise refinement. At each stage of this process a complete problem or subproblem is solved, but the details of the solution are not permitted to grow. Instead, when the solution threatens to become too detailed, parts of it are split off as new subproblems, to be attacked in the next stage. CF Pascal, with comments in English for the deferred details, is an excellent vehicle for expressing the steps of a design. Once a design is complete—that is, when all of its comments have been expressed as executable Pascal text,

the program can be accumulated and executed. However, there are advantages to collecting and testing it in stages. Using the text from only a few design parts focuses attention on any defects that may lurk in the design, and makes the task of transcribing into machine-readable form more manageable. The principles of design and development are summarized in Section 2.3.4.

Procedures are a convenient feature of Pascal to use with stepwise refinement, because they encapsulate and name blocks of Pascal text. Then when it comes time to turn the design into a program, they organize the text as well. Procedures can be reused so that larger and larger problems can be solved without repeating the design effort.

CHAPTER 3

BOOLEAN LOGIC AND CONDITIONAL EXECUTION

Chapter Preview

In this chapter you will learn how to use Boolean logic in designing conditional statements in CF Pascal. The problem of sorting small character strings illustrates a systematic approach to the design of conditional statements. Finally, a new CF Pascal facility for using expressions in Boolean logic is introduced and applied to the sorting problem.

New ideas: sorting a string of characters, finding the minimum character of a string, comparing two program designs.

3.1 Sorting with Conditional Execution

Preview

This section demonstrates how to design and analyze a family of sort programs, using status comments to maintain intellectual control of deeply nested CF Pascal statements. The analysis leads to a new family of better sort programs.

New ideas: status comments, designing a family of programs, improving the design for a problem.

Consider the problem of sorting a character string given in the input file to the output file. The input characters are to be rearranged into ascending alphabetic order for output. If there are duplicate characters in the input, they are to be similarly duplicated in the output. For example:

input string	sorted output string
abc	**abc**
cab	**abc**
character	**aaccehrrt**

The methods to be used for sorting will begin with simple cases and work up to the general problem.

In a discussion of character strings, it is convenient to name them so that the length is part of the name. Thus a string of length n will be called an "n-string". For example, **cab** is a 3-string. In examples, blank characters will not appear unless they are explicitly shown using □.

3.1.1 The IFSort Programs

The simplest sorting problem might involve 1-strings, but that is no problem at all. Any 1-string is already sorted.

For 2-strings a strategy immediately suggests itself: read the two characters, compare them, and write them out in the proper order. Thus, a 2-string can be sorted with a single IF statement:

Design Part 1

```
PROGRAM IFSort2(INPUT,OUTPUT);
{Sort a 2-string from INPUT to OUTPUT}
  VAR
    Ch1, Ch2: CHAR;
  BEGIN {IFSort2}
    READ(Ch1, Ch2);
    WRITELN('INPUT DATA ', Ch1, Ch2);
    WRITE('SORTED DATA ');
    IF Ch1 < Ch2
    THEN {Sort Ch1, Ch2 to OUTPUT,
          knowing that Ch1 < Ch2}
      WRITELN(Ch1, Ch2)
    ELSE {Sort Ch1, Ch2 to OUTPUT,
          knowing that Ch1 >= Ch2}
      WRITELN(Ch2, Ch1)
  END.  {IFSort2}
```

Executions

```
INPUT :ca
OUTPUT:INPUT DATA ca
       SORTED DATA ac
INPUT :37
OUTPUT:INPUT DATA 37
       SORTED DATA 37
```

Since Design Part 1 is fully realized in CF Pascal, it could also be called Development Program 1A. The second line of output is completed by one of two WRITE statements, chosen by the IF statement. The comments in the THEN and ELSE parts describe simpler problems because of the known relationship between the values of Ch1 and Ch2.

The same sorting strategy can be applied to three characters using an IF statement and variables Ch1, Ch2, and Ch3. The development will illustrate the use of comments for intellectual control of design. This time the task cannot be accomplished so easily in a single step, so the design begins with the main task expressed as a comment for further refinement. Only reading and echoing appear as Pascal code at this point:

Design Part 2

```
PROGRAM IFSort3(INPUT,OUTPUT);
{Sort a 3-string from INPUT to OUTPUT}
  VAR
    Ch1, Ch2, Ch3: CHAR;
  BEGIN {IFSort3}
    READ(Ch1, Ch2, Ch3);
    WRITELN('INPUT DATA ', Ch1, Ch2, Ch3);
    WRITE('SORTED DATA ');
    {Sort Ch1, Ch2, Ch3 to OUTPUT}
  END.  {IFSort3}
```

To accomplish the sorting task at the next refinement step, Ch1 and Ch2 are compared first. The choice is an arbitrary one, but it copies IFSort2. The comparisons add knowledge that makes the task remaining simpler:

Design Part 2.1

```
BEGIN {Sort Ch1, Ch2, Ch3 to OUTPUT}
  IF Ch1 < Ch2
  THEN
    {Ch1 < Ch2: Sort Ch1, Ch2, Ch3 to OUTPUT}
  ELSE
    {Ch1 >= Ch2: Sort Ch1, Ch2, Ch3 to OUTPUT}
END
```

The comments describing the remaining tasks give the additional information followed by a colon and the task to perform. At the THEN part, nothing is known about Ch3, so another IF statement is needed to place it, say by comparing with Ch2:

Design Part 2.1.1

```
{Ch1 < Ch2: Sort Ch1, Ch2 to OUTPUT}
IF Ch2 < Ch3
THEN {Ch1 < Ch2 < Ch3}
  WRITELN(Ch1, Ch2, Ch3)
ELSE
  {Ch1 < Ch2, Ch2 >= Ch3: Sort Ch1, Ch2, Ch3 to OUTPUT}
```

The new comments are derived by combining the new knowledge from the inner IF condition with the knowledge from the comment that begins Design Part 2.1.1. For example, the new IF statement guarantees that the value of Ch3 should follow the value of Ch2; but to enter this design part required that the value of Ch2 followed that of Ch1. Hence the comment on the THEN part states the knowledge:

```
{Ch1 < Ch2 < Ch3}
```

which is enough to complete the task by writing the variables in the right order.

At this point, either the ELSE part from 2.1.1 or the ELSE part from 2.1 can be designed, or the problem could be put aside without losing control of the design. Everything known about the task and data is recorded in the comments. In the ELSE part of 2.1.1, Ch2 must be printed last, but we need a comparison between Ch1 and Ch3 to determine their order.

Design Part 2.1.1.1

```
{Ch1 < Ch2, Ch2 >= Ch3: Sort Ch1, Ch2, Ch3 to OUTPUT}
IF Ch1 < Ch3
THEN {Ch1 < Ch3 <= Ch2: Sort Ch1, Ch2, Ch3 to OUTPUT}
  WRITELN(Ch1, Ch3, Ch2)
ELSE {Ch3 <= Ch1 < Ch2: Sort Ch1, Ch2, Ch3 to OUTPUT}
  WRITELN(Ch3, Ch1, Ch2)
```

In this design step enough new information was gained to obtain the proper WRITE statements for all the new cases. Similarly, the ELSE part of 2.1 is:

Design Part 2.1.2

```
{Ch1 >= Ch2: Sort Ch1, Ch2, Ch3 to OUTPUT}
IF Ch1 < Ch3
THEN {Ch2 <= Ch1 < Ch3: Sort Ch1, Ch2, Ch3 to OUTPUT}
  WRITELN(Ch2, Ch1, Ch3)
ELSE
  {Ch2 <= Ch1, Ch3 <= Ch1: Sort Ch1, Ch2, Ch3 to OUTPUT}
```

And the final refinement is:

Design Part 2.1.2.1

```
{Ch2 <= Ch1, Ch3 <= Ch1: Sort Ch1, Ch2, Ch3 to OUTPUT}
IF Ch2 < Ch3
THEN {Ch2 < Ch3 <= Ch1: Sort Ch1, Ch2, Ch3 to OUTPUT}
  WRITELN(Ch2, Ch3, Ch1)
ELSE {Ch3 <= Ch2 <= Ch1: Sort Ch1, Ch2, Ch3 to OUTPUT}
  WRITELN(Ch3, Ch2, Ch1)
```

In every case the comments have the form

```
{what is known: what is to be done}
```

These design parts together are small enough to assemble into a program in a single step. Since "what is to be done" is the same in each case, only the "what is known" comments are retained on the conditions:

Development Program 2A

```
PROGRAM IFSort3(INPUT,OUTPUT);
{Sort a 3-string from INPUT to OUTPUT}
  VAR
    Ch1, Ch2, Ch3: CHAR;
  BEGIN {IFSort3}
    READ(Ch1, Ch2, Ch3);
    WRITELN('INPUT DATA ', Ch1, Ch2, Ch3);
    WRITE('SORTED DATA ');
    BEGIN {Sort Ch1, Ch2, Ch3 to OUTPUT}
      IF Ch1 < Ch2
      THEN {Ch1 < Ch2}
        IF Ch2 < Ch3
        THEN {Ch1 < Ch2 < Ch3}
          WRITELN(Ch1, Ch2, Ch3)
        ELSE {Ch1 < Ch2, Ch3 <= Ch2}
          IF Ch1 < Ch3
          THEN {Ch1 < Ch3 <= Ch2}
            WRITELN(Ch1, Ch3, Ch2)
          ELSE {Ch3 <= Ch1 < Ch2}
            WRITELN(Ch3, Ch1, Ch2)
      ELSE {Ch2 <= Ch1}
        IF Ch1 < Ch3
        THEN {Ch2 <= Ch1 < Ch3}
          WRITELN(Ch2, Ch1, Ch3)
        ELSE {Ch2 <= Ch1, Ch3 <= Ch1}
          IF Ch2 < Ch3
          THEN {Ch2 < Ch3 <= Ch1}
            WRITELN(Ch2, Ch3, Ch1)
          ELSE {Ch3 <= Ch2 <= Ch1}
            WRITELN(Ch3, Ch2, Ch1)
    END
  END. {IFSort3}
```

Execution

```
INPUT :cabaret
OUTPUT:INPUT DATA cab
       SORTED DATA abc
```

Although a 3-string is only half again longer than a 2-string, IFSort3 (32 lines) is considerably longer than IFSort2 (14 lines). A little study shows why. There are 6 WRITE statements in IFSort3, corresponding to the $3 \times 2 \times 1 = 6$ orderings possible among Ch1, Ch2,

and Ch3. This compares with only 2 WRITE statements in IFSort2, corresponding to the $2 \times 1 = 2$ orderings possible among Ch1 and Ch2. The IF statement of IFSort3 is an expansion of the IF statement of IFSort2, so each of the 2 WRITE statments in IFSort2 is replaced by an IF statement containing 3 WRITE statements.

Similarly, if a program IFSort4 were written to sort a 4-string, if would have each of the 6 WRITE statements of IFSort3 replaced by a composite IF statement with 4 WRITE statements. That is, IFSort4 would contain $4 \times 3 \times 2 \times 1 = 24$ WRITE statements. An IFSort5 would contain $5 \times 4 \times 3 \times 2 \times 1 = 120$ WRITE statments, and so on. For a 10-string, $10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 3{,}628{,}800$ WRITE statements would be required. Evidently the sorting problem requires rethinking.

Three fundamental lessons emerge from the IFSort family of programs:

1. Design nested IF statements by expanding THEN or ELSE parts at each step, accumulating knowledge until all the cases are explicit.
2. Critically analyze programs as you write them.
3. Look for a different problem solution when the analysis shows a design to be impractical.

There is also a fundamental problem-solving principle in the IFSort family:

> To solve a problem think of a *set of problems* of increasing difficulty that includes the given one. Then devise a *family of solutions* starting with the simplest problem in the set.

The principle of starting on the simplest member in a set of problems makes it easier to think of the solution, which can be generalized to solve the real problem. If the solution idea is impractical, analysis is easier when the problem is simple. For example, if you started to solve the problem of sorting 10-strings, and decided to use nested IF statements, you might write hundreds of Pascal statements (with considerable pain and confusion) before abandoning the project. By solving the problem for 2- and 3-strings the designs stayed under control, and the analysis showed clearly that the 10-string problem should not be done this way.

3.1.2 The MinSort Programs

Consider a new strategy for sorting three characters with fewer WRITE statements. Find the minimum character among Ch1, Ch2, and Ch3, write it out, and make sure that the remaining values are in Ch1 and Ch2. This reduces the original problem to a simpler one (that of sorting two values in Ch1 and Ch2). If a variable other than Ch3 is written as the minimum, then the value of Ch3 must be moved to the variable whose value was written.

This preliminary analysis suggests the following design:

Design Part 3

```
PROGRAM MinSort3(INPUT,OUTPUT);
{Sort 3-string from INPUT to OUTPUT}
  VAR Ch1, Ch2, Ch3: CHAR;
  BEGIN {MinSort3}
    READ(Ch1, Ch2, Ch3);
    WRITELN('INPUT DATA ', Ch1, Ch2, Ch3);
    WRITE('SORTED DATA ');
    {Write minimum value to OUTPUT,
     remaining values in Ch1 and Ch2};
    {Sort Ch1, Ch2 to OUTPUT};
    WRITELN
  END.  {MinSort3}
```

Design Part 3.1

```
BEGIN {Write minimum value to OUTPUT,
     remaining values in Ch1 and Ch2}
  IF Ch1 < Ch2
  THEN
    {Write minimum of Ch1, Ch3 to OUTPUT,
     move Ch3 to Ch1 if necessary}
    IF Ch1 < Ch3
    THEN
      BEGIN
        WRITE(Ch1);
        Ch1 := Ch3
      END
    ELSE
      WRITE(Ch3)
  ELSE
    {Write minimum of Ch2, Ch3 to OUTPUT,
     move Ch3 to Ch2 if necessary}
    IF Ch2 < Ch3
    THEN
      BEGIN
        WRITE(Ch2);
        Ch2 := Ch3
      END
    ELSE
      WRITE(Ch3)
END
```

(Design Part 3.1, with 25 lines, violates the 5-15 line size rule for a design part, but is of a very regular structure, and the comments show the situation clearly at every point in the design.) In Design Part 3.1 the variables to be sorted after the minimum has been found are always Ch1 and Ch2, arranged with appropriate assignments after the minimum value is printed. When the nested IF statement thus falls through it remains only to order these two.

Design Part 3.2

```
BEGIN {Sort Ch1, Ch2 to OUTPUT}
  IF Ch1 < Ch2
  THEN
    WRITE(Ch1, Ch2)
  ELSE
    WRITE(Ch2, Ch1)
END
```

When assembled into a development program, MinSort3 has 6 WRITE statements, the same as IFSort3. What would a MinSort4 look like? We need only alter Design Part 3 to include the additional variable Ch4. A new design part will be required to find the minimum value in Ch1, Ch2, Ch3, Ch4, write it out, and leave the remaining characters in Ch1, Ch2, and Ch3. Then Design Parts 3.1 and 3.1.1 can be reused to complete the sorting. That is, MinSort4 could write a minimum and reduce the problem to that of MinSort3.

In order to find the minimum value among four variables, an IF statement nested at least three levels deep will be required because the minimum must precede or equal three other variables. In fact, three levels are sufficient because the conditions of the IF statements can be organized as shown in the table below.

Comparisons to Find a Minimum Among Four Variables

Condition	Minimum
`Ch1 < Ch2`	
`  Ch1 < Ch3`	
`  Ch1 < Ch4`	`Ch1`
`  Ch4 <= Ch1`	`Ch4`
`  Ch3 <= Ch1`	
`  Ch3 < Ch4`	`Ch3`
`  Ch4 <= Ch3`	`Ch4`
`Ch2 <= Ch1`	
`  Ch2 < Ch3`	
`  Ch2 < Ch4`	`Ch2`
`  Ch4 <= Ch2`	`Ch4`
`  Ch3 <= Ch2`	
`  Ch3 < Ch4`	`Ch3`
`  Ch4 <= Ch3`	`Ch4`

The three-level IF statement will need $2 \times 2 \times 2 = 8$ WRITE statements, so the number of WRITE statements in `MinSort4` will be $8 + 4 + 2 = 14$, compared to $4 \times 3 \times 2 \times 1 = 24$ for `IFSort4`.

The table below compares the number of WRITE statements in the two program families:

$2 + 4 + 8 + 16 + \ldots + 2^{N-1}$ (`MinSortN`)

$2 \times 3 \times 4 \times 5 \times \ldots \times N$ (`IFSortN`)

Number of WRITE Statements

N	`IFSortN`	`MinSortN`
3	6	6
4	24	14
5	120	30
6	720	62
7	5,040	126
8	40,320	254
9	362,880	510
10	3,628,800	1,022

A change in design has lead to much smaller programs. However, even the `MinSort` programs quickly grow ever larger.

The `MinSort` family of programs was discovered using another design strategy:

To solve one problem in a family, think in terms of solutions to smaller problems in the family.

The `IfSort` approach transforms solutions for smaller problems in the family into those for larger problems; in contrast, `MinSort` goes in the opposite direction.

In analyzing these simple sort strategies two fundamental principles of program design appear:

1. Thinking harder about a design can lead to a smaller program.
2. A good design strategy is to sove a problem by reducing it to a simpler case.

3.1.3 Exercises

3.1.1 Check the syntax correctness of the main IF statement in `IFSort2` by constructing an abbreviated generation tree.

3.1.2 Construct an execution table for Design Part 1 `IFSort2` on input **`XT`**.

3.1.3 Write a program named MySort2 that sorts a string of two characters in the format:

```
INPUT :ca
OUTPUT:THE INPUT (c, a) SORTS INTO (a, c)
```

Will several design steps be required?

3.1.4 Consider the following portion that might be part of IFSort4:

```
THEN {Ch1 < Ch2 < Ch3: Sort}
  IF Ch4 < Ch2
  THEN {Ch1 < Ch2 < Ch3, Ch4 < Ch2: Sort}
    IF Ch4 < Ch1
    THEN {Ch4 < Ch1 < Ch2 < Ch3}
      {Ch4 < Ch1 < Ch2 < Ch3: Sort Ch1, Ch2, Ch3, Ch4}
    ELSE {Ch1 <= Ch4 < Ch2 < Ch3}
      {Ch1 <= Ch4 < Ch2 < Ch3: Sort Ch1, Ch2, Ch3, Ch4}
  ELSE
    {Ch1 < Ch2 < Ch3, Ch2 <= Ch4: Sort Ch1, Ch2, Ch3, Ch4}
```

Including any needed comments, fully expand the THEN and ELSE parts that remain.

3.1.5 Write the IF statement that prints the minimum of the contents of four variables using eight WRITE statements, as described in Section 3.1.2.

3.1.6 A file is "almost sorted" if all its characters are in order except for one. That is, sorting requires only removing one character and inserting it elsewhere in the file.

a) Design and develop a program using IF statements to sort a 3-string that is almost sorted.

b) Repeat a) for a 4-string, but do only the design.

c) How many WRITE statements will be needed for almost sorted 3-strings, 4-strings, ..., 10-strings?

3.2 Boolean Logic

Preview

Boolean logic provides a formal and systematic method for reasoning about data conditions in IF statements. The method simplifies analysis of program behavior.

New ideas: truth values, Boolean values, NOT, AND, OR Boolean operators, Boolean expressions, truth tables, Boolean identities.

The Pascal IF statement and WHILE statement provide for conditional execution, depending on the outcome of a <condition> during execution. For example, the statement

```
IF Ch1 < Ch2
THEN
  WRITE(Ch1)
ELSE
  WRITE(Ch2)
```

executes the THEN part or the ELSE part, depending on whether or not the value of Ch1 precedes the value of Ch2 at that point in execution. Such a comparison has only two possible outcomes—it holds or it doesn't hold. Conditions with just two possible outcomes are called *Boolean conditions* (after George Boole, 1815-1864, who was one of the first people to study this subject in a systematic way). The outcomes are called *truth values,* or *Boolean values,* with the names TRUE and FALSE. These are the values taken by a <condition> in an IF or WHILE statement, just as character variables take character values. For example, immediately after the assignments

```
Ch1 := 'A';
Ch2 := 'C'
```

some Boolean values are:

<condition>	Boolean value
`Ch1 < Ch2`	TRUE
`Ch1 >= Ch2`	FALSE
`Ch1 <= Ch2`	TRUE
`Ch1 = 'B'`	FALSE
`Ch2 = 'C'`	TRUE
`'X' = 'Y'`	FALSE

Conditions and Assertions

The IF statement above can be given status comments:

```
IF Ch1 < Ch2
THEN {Ch1 < Ch2}
  WRITE(Ch1)
ELSE {Ch1 >= Ch2}
  WRITE(Ch2)
```

The <condition> `Ch1 < Ch2` on the first line may have value TRUE or may have value FALSE, depending on the values of `Ch1` and `Ch2`. However, the THEN status comment `{Ch1 < Ch2}` on the next line can be thought of as an *assertion* that the value of the <condition> must be TRUE on entry to the THEN part of the IF statement. The basis for this assertion is an understanding of how Pascal IF statements execute. That is, no matter what has happened before, no matter what the values of `Ch1` and `Ch2` may be, upon entry to the THEN part (and every time the THEN part is entered) the value of `Ch1 < Ch2` must be TRUE.

3.2.1 Boolean Operators

The Operator NOT

The IF statement above illustrates how to reason about the status of variable values in a program. The THEN-part status comment asserts that the value of `Ch1 < Ch2` is TRUE. The ELSE comment reflects our knowledge that the `>=` operator is the opposite of the `<` operator, and FALSE is defined to be the opposite of TRUE. That is, the value of `Ch1 >= Ch2` is TRUE when the value of `Ch1 < Ch2` is FALSE and vice versa. The Boolean operator NOT reverses a truth value. That is:

NOT(TRUE) = FALSE
NOT(FALSE) = TRUE

NOT is a *unary* (applies to a single operand), *prefix* (preceding its operand) operator.

Since

NOT (NOT (FALSE)) = NOT (TRUE) = FALSE
NOT (NOT (TRUE)) = NOT (FALSE) = TRUE

and these are the only possibilities,

NOT (NOT (*P*)) = *P*

for any Boolean value *P*. An identity of this kind expresses a fact for all possible Boolean values *P* in a single formula.

The Operator AND

Consider a nested IF statement with status comments:

```
IF Ch1 < Ch2
THEN {Ch1 < Ch2}
  IF Ch2 < Ch3
  THEN {Ch1 < Ch2 < Ch3}
    WRITE(Ch1)
  ELSE {Ch1 < Ch2, Ch3 <= Ch2}
    IF Ch1 < Ch3
    THEN {Ch1 < Ch3 <= Ch2}
      WRITE(Ch1)
    ELSE {Ch3 <= Ch1 < Ch2}
      WRITE(Ch3)
```

The status comment

```
{Ch1 < Ch2 < Ch3}
```

expresses the two assertions that the value of `Ch1` precedes that of `Ch2`, and the value of `Ch2` precedes that of `Ch3`. The Boolean operator AND captures this intuitive meaning. AND is a *binary* (two operands) *infix* (placed between its operands) operator that takes the value TRUE if the values of both Boolean operands are TRUE; for all other Boolean values of the operands, it takes the value FALSE. That is:

TRUE AND TRUE = TRUE
TRUE AND FALSE = FALSE
FALSE AND TRUE = FALSE
FALSE AND FALSE = FALSE

Comparison operations can be usefully combined with AND operator. When the same operands appear in a combined condition, there is often a simpler condition with the same truth value. For example:

Comparisons combined with AND		Simpler equivalent comparison
`Ch1 <= Ch2`	`Ch1 >= Ch2`	`Ch1 = Ch2`
`Ch1 <= Ch2`	`Ch1 < Ch2`	`Ch1 < Ch2`
`Ch1 < Ch2`	`Ch1 > Ch2`	FALSE

The simpler comparison is equivalent because AND requires both of the combined comparisons to hold, precluding possibilities that are contradictory. In the second case, for example, equality cannot hold in `Ch1 <= Ch2` because `Ch1 < Ch2` forbids it.

The Operator OR

In the preceding nested IF statement, there are two ways that a `WRITE(Ch1)` can be reached. The value of `Ch1 < Ch2` must be TRUE and one or both of the values of `Ch2 < Ch3`, `Ch1 < Ch3` must be TRUE. (If both these values are TRUE, it is the first WRITE statement that is reached.) The Boolean operator OR is a *binary infix* operator that captures this intuitive meaning:

TRUE OR TRUE = TRUE
TRUE OR FALSE = TRUE
FALSE OR TRUE = TRUE
FALSE OR FALSE = FALSE

Here are some useful combinations using OR:

Comparisons combined with OR		Simpler equivalent comparison
`Ch1 < Ch2`	`Ch1 > Ch2`	`Ch1 <> Ch2`
`Ch1 <= Ch2`	`Ch1 < Ch2`	`Ch1 <= Ch2`
`Ch1 <= Ch2`	`Ch1 >= Ch2`	`TRUE`

In OR combinations, only one of the combined conditions need hold, so the equivalent is weaker than either. For example, in the first case, one value is less than the other, but since it could be either one, all that can be concluded is that they are not the same.

Truth Tables for Boolean Operators

The Boolean operators are defined by their actions on the two Boolean values TRUE and FALSE. The *truth table* for a Boolean operator is a summary of this definition, giving in orderly rows all the possible combinations of Boolean values and the outcome of the operation on those values. In truth tables, TRUE is usually abbreviated T and FALSE abbreviated F.

Truth Tables for Boolean Operators

P	*Q*	NOT *P*	*P* AND *Q*	*P* OR *Q*
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

It is a good idea to memorize and use these definitions when dealing with NOT, AND, and OR. Sometimes it is safe to reason using the intuitive ideas of negation (not), combination (and), and alternation (or), but not always. For example, in the "exclusive" meaning of "or", "one *or* the other" means that one may be true, the other may be true, but they are not *both* true. As the truth table shows, this is *not* the meaning of OR.

3.2.2 Boolean Expressions

Boolean operators and operands can be combined with parentheses into *Boolean expressions*. Any such expression whose innermost operands are Boolean values can be transformed to a single truth value using the truth table definitions. For example, using T and F for the truth values,

T OR ((NOT F) AND F) = T.

The result can be obtained one step at a time by repeatedly finding an innermost parenthesized expression and replacing it with its value from the truth table. In this case, the successive expressions are:

Expression	*Reduction Step*	*Reason*
T OR ((NOT F) AND F)	(NOT F) = T	NOT.3
T OR (T AND F)	(T AND F) = F	AND.2
T OR F	(T OR F) = T	OR.2
T		

so the expression reduces to T. The "Reason" column identifies the table and line used to justify each step.

3.2.3 Boolean Identities

An equation between two Boolean expressions that holds no matter what Boolean values are consistently substituted, such as

NOT (NOT P) = P

is called a *Boolean identity.*

Fifteen useful Boolean identities are given below:

1. TRUE AND $P = P$
2. TRUE OR P = TRUE
3. FALSE AND P = FALSE
4. FALSE OR $P = P$
5. NOT (NOT P) = P
6. P AND (NOT P) = FALSE
7. P OR (NOT P) = TRUE
8. P OR $Q = Q$ OR P
9. P AND $Q = Q$ AND P
10. P OR (Q OR R) = (P OR Q) OR R
11. P AND (Q AND R) = (P AND Q) AND R)
12. P OR (Q AND R) = (P OR Q) AND (P OR R)
13. P AND (Q OR R) = (P AND Q) OR (P AND R)
14. NOT (P OR Q) = (NOT P) AND (NOT Q)
15. NOT (P AND Q) = (NOT P) OR (NOT Q)

Proofs of Boolean identities can be organized as tables in which all the combinations of operand Boolean values are systematically listed, and each part of the identity is evaluated from the basic truth tables, then combined to find the value of the left and right sides. If those values are the same for all cases, the identity is established. Here are proofs of some identities:

1. TRUE AND $P = P$

P	Left	Right
T	T	T
F	F	F

8. P OR $Q = Q$ OR P

P	Q	Left	Right
T	T	T	T
T	F	T	T
F	T	T	T
F	F	F	F

12 P OR (Q AND R) = (P OR Q) AND (P OR Q)

P	Q	R	Q AND R	P OR Q	P OR R	Left	Right
T	T	T	T	T	T	T	T
T	T	F	F	T	T	T	T
T	F	T	F	T	T	T	T
T	F	F	F	T	T	T	T
F	T	T	T	T	T	T	T
F	T	F	F	T	F	F	F
F	F	T	F	F	T	F	F
F	F	F	F	F	F	F	F

Identities 8 and 9 are called ***commutativity*** identities. They can be used to reverse terms and get new identities, for example from Identity 4:

P OR FALSE = P.

Identities 10 and 11 are called ***associativity*** identities. They permit expressions to be written without parentheses, since the results are identical whether operations are associated to the left or to the right. Identities 12 and 13 are called ***distributivity*** identities. They can usefully be reversed, using the commutativity identities, to get (for example from 13):

(P OR Q) AND R = (P AND R) OR (Q AND R)

Identities 14 and 15, called ***de Morgan's laws,*** are the least obvious.

These identities are useful tools that should be understood and available for reference and use in program design and analysis. Their use can be illustrated by simplifying the expression

P OR ((NOT P) AND Q).

In this case,

P OR ((NOT P) AND Q)
= (P OR (NOT P)) AND (P OR Q) by Identity 12
= TRUE AND (P OR Q) by Identity 9
= P OR Q by Identity 1

a much simpler expression.

3.2.4 Exercises

3.2.1 Calculate the value of each expression, giving each reduction and its reason in tabular form:

a) T OR (F AND (T AND (NOT F)))
b) (T AND (NOT F)) AND (F OR F)
c) NOT ((T AND (T OR T)))
d) F AND (T OR (F OR (NOT T)))
e) (F OR (NOT T)) OR (T OR T)

3.2.2 Determine the validity of the following Boolean identities by using truth tables.

a) P AND (P OR Q) = P AND Q
b) NOT (P AND (Q AND R)) = NOT ((P AND R) AND Q)
c) NOT (P OR Q) = (NOT P) AND (NOT (Q OR P))
d) P OR (Q AND R) = (R AND Q) OR P
e) P OR (NOT (Q OR R)) = P AND (Q AND R)

3.2.3 Prove each of the following from Section 3.2.3:

a) identity 4

b) identity 7

c) identity 10

d) identity 14

e) identity 15

3.2.4 There are additional Boolean operators in common use. For example, IMPLIES is defined by the truth table

P	Q	P IMPLIES Q
T	T	T
T	F	F
F	T	T
F	F	T

Write a Boolean expression β such that

$\beta = P$ IMPLIES Q

using only AND, OR, NOT, P, Q in β.

3.2.5 Using the definition of IMPLIES from the previous exercise, define the Boolean operator EQUIV as

P EQUIV $Q = (P$ IMPLIES $Q)$ AND $(Q$ IMPLIES $P)$

Find a Boolean expression equivalent to P EQUIV Q using only AND, OR, NOT, P, Q. (Hint: work out a truth table for EQUIV first.)

3.3 Boolean Conditions in CF Pascal

Preview

Boolean expressions can be included in CF Pascal IF statements and WHILE statements to reduce the complexity of many programs. In illustration, MinSort is revised, using Boolean conditions in place of nested IF statements where possible.

New ideas: Pascal Boolean expressions

NOT, AND, OR are standard words in CF Pascal, and Boolean expressions made with them can be used in the <condition> of an IF or WHILE statement. That is, Syntax Rule 23 of Chapter 2 is altered to the following:

SR23. <condition> ::= <expression> <comparison> <expression>
¦ NOT (<condition>)
¦ (<condition>) AND (<condition>)
¦ (<condition>) OR (<condition>)

The rule shows that any <condition> used with a Boolean operator must be surrounded by parentheses.

The meanings of Pascal <condition>s are the counterparts of the formal Boolean expressions introduced in Section 3.2.2. For example, the value of

```
NOT(Ch1 < Ch2)
```

is FALSE if and only if the value of `Ch1 < Ch2` is TRUE, and vice versa. That is, if B is any Pascal Boolean expression, the value of

```
NOT( B )
```

and the value of *B* itself are opposites—one is NOT the other. The CF Pascal <comparison> operators also come in opposite pairs. For example, the value of

```
Ch1 = Ch2
```

and that of

```
Ch1 <> Ch2
```

are opposites. Hence applying the Pascal NOT operator to either of these expressions will yield the same Boolean value as the other. For example, the expressions

```
NOT(Ch1 = Ch2)
```

and

```
Ch1 <> Ch2
```

will always have the same Boolean value, and hence one expression can be used in place of the other. Examples of the interchangeable combinations are:

Expressions with the Same Value	
`NOT(Ch1 < Ch2)`	`Ch1 >= Ch2`
`NOT(Ch1 > Ch2)`	`Ch1 <= Ch2`
`NOT(Ch1 = Ch2)`	`Ch1 <> Ch2`

The AND operator also combines with the <comparison>s to yield some useful equivalents:

Expressions with the Same Value	
`(Ch1 <= Ch2) AND (Ch1 >= Ch2)`	`Ch1 = Ch2`
`(Ch1 <= Ch2) AND (Ch1 < Ch2)`	`Ch1 < Ch2`
`(Ch1 <> Ch2) AND (Ch1 > Ch2)`	`Ch1 > Ch2`

One combination,

```
(Ch1 < Ch2) AND (Ch1 > Ch2)
```

is usually only written by mistake, because it always takes the value FALSE. It is equivalent to many other expressions, for example `'A' = 'B'`, but those would probably not be written either.

The OR operator has the following useful equivalents:

Expressions with the Same Value	
`(Ch1 < Ch2) OR (Ch1 > Ch2)`	`Ch1 <> Ch2`
`(Ch1 <= Ch2) OR (Ch1 < Ch2)`	`Ch1 <= Ch2`

The expression

```
(Ch1 = Ch2) OR (Ch1 <> Ch2)
```

is probably only written by mistake, since it always takes the value TRUE, as for example does `'0' <> '1'`.

Boolean identities and facts can be used to create equivalent Pascal <condition>s. For example, de Morgan's law

NOT(*P* OR *Q*) = (NOT *P*) AND (NOT *Q*)

along with the combination properties of <comparison>s means that IF statements beginning

```
IF NOT((Ch1 = Ch2) OR (Ch3 < Ch4))
```

and

```
IF (Ch1 <> Ch2) AND (Ch3 >= Ch4)
```

will have identical actions; most people find the second one easier to understand.

Boolean expressions are very useful in programming. For example,

```
WHILE (Ch = ' ') OR (Ch = '0')
DO
  READ(Ch)
```

will flush to the first nonblank, nonzero character.

```
IF (Ch >= '0') AND (Ch <= '9')
THEN
  WRITE(Ch)
```

will write only digits.

3.3.1 The MinSort Programs Again

Boolean expressions allow the `MinSort` programs to be improved, creating a new family `MinSortB`. Design Part 3.1 can be replaced by the following:

Design Part 3.1′

```
BEGIN {Write minimum value of Ch1, Ch2, Ch3
       to OUTPUT, other values in Ch1, Ch2}
  IF (Ch1 < Ch2) AND (Ch1 < Ch3)
  THEN
    {Write minimum value of Ch1, Ch3 to OUTPUT,
     moving Ch3 to Ch1 if necessary}
    BEGIN
      WRITE(Ch1);
      Ch1 := Ch3
    END
  ELSE
    {Write minimum value of Ch2, Ch3 to OUTPUT,
     moving Ch3 to Ch2 if necessary}
    IF Ch2 < Ch3
    THEN
      BEGIN
        WRITE(Ch2);
        Ch2 := Ch3
      END
    ELSE
      WRITE(Ch3)
END
```

Design Part 3.1′ (say it is for one of a new family called `MinSort3B`) contains 3 WRITE statements instead of 4, so using it, `MinSort3B` needs only $3 + 2 = 5$ WRITE statements. Looking on to `MinSort4B`, the table below shows an organization of Boolean expressions in which a minimum can be found among `Ch1`, `Ch2`, `Ch3`, and `Ch4` and written using only four WRITE statements.

Comparisons to Find a Minimum Among Four Variables

Expression	Minimum
`(Ch1 < Ch2) AND (Ch1 < Ch3) AND (Ch1 < Ch4)`	`Ch1`
`(Ch2 < Ch3) AND (Ch2 < Ch4)`	`Ch2`
`Ch3 < Ch4`	`Ch3`
`{else}`	`Ch4`

Therefore, the number of WRITE statements in `MinSort4B` need only be $4 + 3 + 2 = 9$. In general, the number of WRITE statements for sorting an n-string is

$$n + (n - 1) + \ldots + 3 + 2$$

The comparison with `IFSort` and `MinSort` is:

Number of WRITE Statements

N	IFSortN	MinSortN	MinSortNB
3	6	6	5
4	24	14	9
5	120	30	14
6	720	62	20
7	5,040	126	27
8	40,320	254	35
9	362,880	510	44
10	3,628,800	1,022	54

The table tells a dramatic tale. The first reduction results from a better idea on how to sort. The second reduction results from a more powerful language facility in CF Pascal. However, to sort a 1000-string using `MinSort1000B` will still require 500,499 WRITE statements.

3.3.2 Exercises

3.3.1 Give a syntax tree for the <IF statement>

```
IF (Ch1 = 'A') AND ((Ch2 <> Ch1) OR (Ch1 < 'H'))
THEN
  Ch1 := 'Q'
```

abbreviating all but the structure of the Boolean expression.

3.3.2 Show that

```
NOT(Ch1 <> Ch1) AND (Ch1 = Ch1)
```

is not a <condition>. Give two other <condition>s that the person who wrote this might have meant, and abbreviated generation trees for them.

3.3.3 The syntax rule that requires parentheses in Pascal Boolean expressions has been said to be good because it forces programmers to make their meaning clear in an area where people have difficulty understanding. Give a counterargument based on the string

```
IF Ch1 = 'A' OR Ch1 = 'B' OR Ch1 = 'C'
THEN
  Ch1 := ' '
```

and facts about Boolean expressions not in Pascal.

3.3.4 When `X1` and `X2` have character values, do the two program fragments

```
IF X1 = 'A'
THEN
  IF X2 = 'B'
  THEN
    S
```

and

```
IF (X1 = 'A') AND (X2 = 'B')
THEN
  S
```

have exactly the same effect, where *S* is a statement?

3.3.5 When `X1` and `X2` have character values, do the program fragments

```
IF X1 = 'A'
THEN
  S;
IF X1 = 'B'
THEN
  S
```

and

```
IF X1 = 'A' OR X1 = 'B'
THEN
  S
```

have the same effect, where *S* is a statement? (Hint: it matters what *S* does.)

3.3.6 In the `SarahRevere` program of Section 2.3.3, one might imagine combining the entire test for **`land`** or **`sea`** into one <condition>:

```
IF (W1 = 'l') AND (W2 = 'a' OR W2 = 's') AND
   (W3 = 'n' OR W3 = 'e') AND (W4 = 'd' OR W4 = 'a')
THEN {either 'land' or 'sea' has been found}
  ...
```

Criticize the syntax and the semantics of this idea.

3.3.7 The Boolean identities of Section 3.2.3 can be written in a Pascal-like syntax. For example, number 10 would be

```
P OR (Q OR R) = (P OR Q) OR R
```

a) Using the syntax for CF Pascal Boolean expressions, and a suitable definition for <Boolean variable> that allows any uppercase letter between `P` and `Z`, write complete syntax equations for Boolean identities.

b) The syntax of a) will permit things like

```
P OR Q = P AND Q
```

that are *not* Boolean identities. Give a context rule that along with a) allows only the forms that actually are identities.

3.4 Chapter Summary

Sorting using nested IF statements illustrates the problem-solving principle of generalizing a given problem to a set of problems, and working up from its simplest member. Nested IF statements can be difficult to keep under intellectual control. The status comment, a device to record the relationship between variable values at each point in the computation, makes designs straightforward even for deep nesting.

Status comments are an example of the use of Boolean logic to reason about the actions of programs. Whenever complex situations involving negations, combinations, alternatives arise, the Boolean operators NOT, AND, OR make formal analysis possible. Boolean expressions can always be understood by constructing truth tables, but many facts about Boolean conditions can be best expressed in Boolean identities, and properties of the <comparison> operators.

Pascal has `NOT`, `AND`, `OR` operators corresponding to those of Boolean logic, which can be used to simplify IF statements and WHILE statements. When seeking an appropriate Pascal Boolean expression, it is often helpful to use facts from formal Boolean logic.

CHAPTER 4

TEXT FILES AND ITERATIVE EXECUTION

Chapter Preview

In this chapter you will learn how to use a new Pascal facility, the TEXT file. Thus far, the only variables available to hold working data are character variables. With TEXT files, working data can be temporarily stored without regard for its volume. Reading and writing TEXT files allows the sorting of strings of arbitrary length.

4.1 TEXT Files

Preview

A new Pascal data type, the TEXT file, is introduced to hold working data during program execution. TEXT files can hold any number of characters in a sequence, for examination one character at a time.

New ideas: TEXT file data type, using files to store working data, REWRITE, RESET.

4.1.1 File Operations

INPUT and OUTPUT are standard identifiers naming files that can be read and written; these files are sequences of characters. Pascal also has variables whose values are sequences of characters—TEXT files. (The TEXT—FILE OF CHAR—file is the other data type in Character and File Pascal.) TEXT files may be created, written, and read back repeatedly, and TEXT file variables can contain multiple characters, as CHAR variables cannot. Files are declared in the VAR section using the standard Pascal word TEXT instead of CHAR. For example:

```
VAR
  Ch:  CHAR;
  Chars:  TEXT;
```

Once declared, a file must be prepared for writing with a REWRITE statement, in this case:

```
REWRITE(Chars)
```

Or a file can be prepared for reading (after some data have been written to it) by a RESET statement, in this case:

```
RESET(Chars)
```

If a REWRITE statement has been executed for a file and there are no intervening RESET statements, the file is open for writing and WRITE statements may refer to it, here for example:

```
WRITE(Chars, Ch, '#');
WRITELN(Chars, Ch, Ch)
```

If a RESET statement has been executed for a file with no intervening REWRITE statements, then the file is open for reading and READ statements may refer to it, as in:

```
READ(Chars, Ch)
```

Reading and writing programmer-declared files are similar in every respect to reading from INPUT and writing to OUTPUT. The value of a file that is opened for reading or writing can be thought of as a character string and a cursor. The REWRITE statement sets the cursor at the first place of a string of no characters, just as OUTPUT is set by the program header. The RESET

statement preserves the data of the file and moves the cursor to the first character (if any), just as INPUT is set by the program header.

OUTPUT is always open for writing, and OUTPUT may be the file name in a WRITE statement, so that, for example,

```
WRITE (Ch)
```

and

```
WRITE (OUTPUT, Ch)
```

have the same effect. Similarly, INPUT is always open for reading, and

```
READ (INPUT, Ch)
```

may be used.

4.1.2 File Syntax and Semantics

To incorporate file operations into the syntax and context rules of CF Pascal requires changes in Syntax Rules 7, 12, and 13, and an additional Rule 29:

SR7. <declarations> ::= VAR <identifier list> : <type>
¦ <declarations> ; <identifier list> : <type>

SR12. <READ statement> ::= READ (<identifier list>)
¦ RESET (<identifier>)

SR13. <WRITE statement> ::= WRITE (<write list>)
¦ WRITELN (<write list>)
¦ WRITELN
¦ REWRITE (<identifier>)

SR29. <type> ::= CHAR ¦ TEXT

The corresponding context rule is:

CR6. A TEXT identifier is an <identifier> declared to be of <type> TEXT. If a TEXT identifier appears in a <READ statement> or a <WRITE statement>, it must be the first <identifier>. A TEXT identifier, other than INPUT or OUTPUT, that appears in a READ statement or WRITE statement must appear in <declarations>. Only TEXT identifiers other than INPUT or OUTPUT may appear in RESET or REWRITE statements.

Variables declared as TEXT must be properly prepared before READ and WRITE statements are applied to them. The first operation must be a REWRITE statement, which makes the file ready for writing. Until the REWRITE statement, the file has an unknown value, just as an uninitialized CHAR variable does before assignment. A file may have REWRITE statements applied to it repeatedly, each time beginning it with an empty string. The last write operation applied to a file should be a WRITELN statement; then the file is ready for a RESET statement, which makes it ready for reading. The characters previously written following REWRITE may then be read in order following RESET. No file may be read beyond what has been written into it; to do so will cause the program to abort in execution. However, the line marker that the final WRITELN attaches to a file will be read as a blank; then on the *next* READ statement the program will abort. A file may be RESET many times.

4.1.3 Copying Files

A typical file operation copies one file to another. Program CopyTwice copies INPUT to OUTPUT through an intermediate working file Chars. The first step is to copy INPUT to Chars; the second step is to copy Chars to OUTPUT.

Design Part 1

```
PROGRAM CopyTwice(INPUT,OUTPUT);
  {Copy characters preceding # from INPUT to OUTPUT.
   The program will abort if INPUT contains no #.}
  VAR
    Ch:  CHAR;
    Chars:  TEXT; {Working file}
  BEGIN {CopyTwice}
    {Copy INPUT to Chars};
    {Copy Chars to OUTPUT}
  END.  {CopyTwice}
```

Design Part 1.1

```
BEGIN {Copy INPUT to Chars}
  REWRITE(Chars); {Prepare Chars for writing}
  READ(INPUT, Ch);
  WHILE Ch <> '#'
  DO
    BEGIN {Copy a single char to Chars}
      WRITE(Chars, Ch);
      READ(INPUT, Ch)
    END;
  WRITELN(Chars, '#') {Close out Chars with an end marker}
END;
```

Design Part 1.2

```
BEGIN {Copy Chars to OUTPUT}
  RESET(Chars); {Prepare Chars for reading}
  READ(Chars, Ch);
  WHILE Ch <> '#'
  DO
    BEGIN
      WRITE(OUTPUT, Ch);
      READ(Chars, Ch)
    END;
  WRITELN(OUTPUT)
END
```

Without the # end marker that Design Part 1.1 puts on the intermediate file, Design Part 1.2 would not be able to stop its iteration. (And since # was the input end marker, it is known to be safe to use.)

These design parts are small enough to assemble into a program at one step.

Development Program 1A

```
PROGRAM CopyTwice(INPUT,OUTPUT);
  {Copy characters preceding # from INPUT to OUTPUT.
   The program will abort if INPUT contains no #.}
  VAR
    Ch:  CHAR;
    Chars:  TEXT; {Working file}
  BEGIN {CopyTwice}
```

```
    BEGIN {Copy INPUT to Chars}
      REWRITE(Chars); {Prepare Chars for writing}
      READ(INPUT, Ch);
      WHILE Ch <> '#'
      DO
        BEGIN
          WRITE(Chars, Ch);
          READ(INPUT, Ch)
        END;
      WRITELN(Chars, '#') {Close out Chars with an end marker}
    END;
    BEGIN {Copy Chars to OUTPUT}
      RESET(Chars); {Prepare Chars for reading}
      READ(Chars, Ch);
      WHILE Ch <> '#'
      DO
        BEGIN
          WRITE(OUTPUT, Ch);
          READ(Chars, Ch)
        END;
      WRITELN(OUTPUT)
    END
  END.  {CopyTwice}
```

Executions

```
INPUT :ABCD#
OUTPUT:ABCD

INPUT :1#ABCD
OUTPUT:1

INPUT :Four score and seven years ago#
OUTPUT:Four score and seven years ago
```

The comment describing the behavior of CopyTwice

```
{Copy characters preceding # from INPUT to OUTPUT.
 The program will abort if INPUT contains no #.}
```

says nothing about the file Chars. The comment for the entire program should describe only the behavior that can be observed from outside, because it is intended for someone who wishes to use the program rather than understand how it works. Later comments apply to the internal details. This generalizes to a principle for comments:

Use variable names in comments only after those names have been declared.

Observing this rule will automatically keep information about *how* a program works out of comments intended only to say *what* it does.

The partial execution table below shows the contents of the files during execution.

Partial Execution Table for Development Program 1A

After Execution of	Condition	INPUT	OUTPUT	Chars	Ch
REWRITE(Chars)		AB#	_	_	?
READ(INPUT,Ch)		AB#	_	_	A
WHILE Ch<>'#'	TRUE	AB#	_	_	A
BEGIN		AB#	_	_	A
WRITE(Chars,Ch)		AB#	_	A_	A
READ(INPUT,Ch)		AB#	_	A_	B
END		AB#	_	A_	B
WHILE Ch<>'#'	TRUE	AB#	_	A_	B
BEGIN		AB#	_	A_	B
WRITE(Chars,Ch)		AB#	_	AB_	B
READ(INPUT,Ch)		AB#_	_	AB_	#
END		AB#_	_	AB_	#
WHILE Ch<>'#'	FALSE	AB#_	_	AB_	#
WRITELN(Chars,'#')		AB#_	_	AB#/_	#
RESET(Chars)		AB#_	_	AB#/	#
READ(Chars,Ch)		AB#_	_	AB#/	A
WHILE Ch<>'#'	TRUE	AB#_	_	AB#/	A
BEGIN		AB#_	_	AB#/	A
WRITE(OUTPUT,Ch)		AB#_	A_	AB#/	A
READ(Chars,Ch)		AB#_	A_	AB#/	B
END		AB#_	A_	AB#/	B
WHILE Ch<>'#'	TRUE	AB#_	A_	AB#/	B
BEGIN		AB#_	A_	AB#/	B
WRITE(OUTPUT,Ch)		AB#_	AB_	AB#/	B
READ(Chars,Ch)		AB#_	AB_	AB#/	#
END		AB#_	AB_	AB#/	#
WHILE Ch<>'#'	FALSE	AB#_	AB_	AB#/	#
WRITELN		AB#_	AB/_	AB#/	#
END.		AB#	AB		

4.1.4 Splitting Files

Consider the problem of splitting INPUT into its odd-numbered and even-numbered members and then printing the odd-numbered members followed by the even-numbered members. A switch can keep track of the odd- and even-numbered members, which can then be copied to files Odds and Evens, respectively. Then these two files can be copied in order to OUTPUT. Program Split carries out this strategy.

Design Part 2

```
PROGRAM Split(INPUT,OUTPUT);
  {Copy INPUT to OUTPUT, characters in odd-numbered
   positions first, followed by those in
   even-numbered positions.  Copying stops at
   the first  #, which must be present.}
VAR
  Ch, Next:  CHAR;
  Odds, Evens:  TEXT;
BEGIN {Split}
  {Split INPUT between Odds and Evens};
  {Copy Odds to OUTPUT};
  {Copy Evens to OUTPUT};
  WRITELN
END.  {Split}
```

`Next` is the flip-flop that records the disposition of the next character read:

Design Part 2.1

```
BEGIN {Split INPUT between Odds and Evens}
  REWRITE(Odds); {Prepare files for writing}
  REWRITE(Evens);
  Next := '0'; {The first character is odd}
  READ(INPUT, Ch);
  WHILE Ch <> '#'
  DO
    {Write Ch to file chosen by Next, flip Next, read Ch};
  WRITELN(Odds, '#'); {Prepare files for RESET}
  WRITELN(Evens, '#')
END
```

Design Part 2.1.1.

```
BEGIN {Write Ch to file chosen by Next, flip Next, read Ch};
  IF Next = '0'
  THEN
    BEGIN
      WRITE(Odds, Ch);
      Next := 'E'
    END
  ELSE {Next = 'E'}
    BEGIN
      WRITE(Evens, Ch);
      Next := '0'
    END;
  READ(INPUT, Ch)
END
```

The Design Parts:

Design Part 2.2

```
{Copy Odds to OUTPUT}
```

Design Part 2.3

```
{Copy Evens to OUTPUT}
```

are just the same as Design Part 1.2, but with `Odds`, `Evens`, respectively, in place of the file `Chars`.

4.1.5 Exercises

4.1.1 Give a syntax tree that demonstrates the syntactic correctness of `CopyTwice`, using abbreviations to emphasize the READ and WRITE statement structure.

4.1.2 Find out exactly what happens on your Pascal machine when each of the following is attempted with a `TEXT` file variable:

a) `WRITE` without `REWRITE`.

b) `RESET` without previous `WRITELN`.

c) `READ` without `RESET`.

d) `READ` too many characters.

4.1.3 Suppose `INPUT` contains exactly two # characters, and call the character string preceding the first *S*, and that between them *T*, so the input form is

S # T #

(and perhaps another string follows this). Design a program to print first the string *T* and then the string *S* on separate lines. Check your program with an execution table for a very short `INPUT`.

4.1.4 When an intermediate file is used, it is often necessary to examine its contents, should something go wrong. Design a program part that will print the contents of `TEXT` file `BadFile`, which is to start and end in the `RESET` state. Do something clever to indicate where blank characters are in the file.

4.1.5 Design a program called `Join` to print alternate characters from two files (each terminated with #). Be careful to consider the cases that both files are of the same length, and that one file is longer than the other, in which case the leftover items should all be printed at the end.

4.1.6 Assemble Design Parts 2 through 2.3 into a Development Program and test it. How many development steps are appropriate?

4.1.7 Design a program that decides if a given file's length is exactly divisible by 3.

4.1.8 Design a program to create a file whose length is the difference between lengths of two given files, develop and test it.

4.1.9 Write a design part to multiply two file lengths, i.e., generate a file `F3` whose length is the product of the lengths of files `F1` and `F2`.

4.1.10 Write a design part to divide two file lengths, i.e., generate a file `F3` whose length is the integer quotient of the lengths of files `F1` and `F2`, and another `F4` whose length is the remainder.

4.2 Sorting with Iterative Execution

Preview

The power of iteration can simplify the sorting problem for character strings.

New ideas: check for sortedness, sorting by selection, bubble sort.

4.2.1 Program SelectSort

Sorting a string of characters with IF statements as in Chapter 3 requires programs that grow with the length of the string to be sorted, with more variables to hold larger strings. A single program to sort a string of any length will now be designed. The string will be held in a `TEXT` file, using only one variable no matter how long the string may be. The `MinSort` idea—to find and write successively the minimum of the string, the minimum of the remainder of the string, and so on, until the entire string has been written out, will work if a string can be copied from one file to another while recognizing and holding out the minimum. Then the file just created (with the minimum removed) could be used again to remove the next minimum, and this could be repeated until the

entire original string was copied in sorted form to the output file.

At first glance it may seem impossible to copy one file to another and hold out the minimum. A trial minimum can be found as the characters pass by, but how can the final minimum value be held out when it is unknown until all have passed? The strategy is to put the old trial minimum back, to write each old trial minimum to the copied file as it is replaced by a new minimum. The file will be copied out of its original order by this process, because a value will be held, then put back out of its original place. But in sorting it doesn't matter how the file is scrambled so long as no characters are lost, and the minimum held out.

How many files will be needed for repeatedly copying and selecting a minimum? If `INPUT` is copied to a working file, say `F1`, with the minimum character held out in `Min`, then `Min` can be printed, and `F1` treated as `INPUT` was. But a new working file, say `F2`, would be needed to hold the next copy, and then another `F3`, and so on, as many files as there are passes over the original data. Since one character is removed from the data at each pass, the number of files needed will be the same as the length of the input. The resulting program could only sort strings of length N if it has N file variables, and would not realize the goal of a sort program for all strings.

If a sequence of files were created as suggested, only the last two would be used: once the data had been copied out of (say) `F5` to `F6`, `F5` (and the others down to `F1` and `INPUT` as well) will never be needed again. This suggests that the intermediate files be reused, in the following scheme:

```
{Copy INPUT to F1}
WHILE {F1 contains data}
DO
  BEGIN
    {Print minimum character from F1};
    {Copy other characters of F1 to F2};
    {Copy F2 to F1}
  END
```

Since `F1` will have one less character each iteration (the one printed and not copied), this process will eventually terminate with `INPUT` printed in sorted order.

Here is the design based on these ideas:

Design Part 3

```
PROGRAM SelectSort(INPUT,OUTPUT);
  {Sort the characters preceding the first  #
   from INPUT to OUTPUT.  The program aborts
   if INPUT contains no  #.}
  VAR
    Ch, Min:  CHAR;
    F1, F2:  TEXT;
  BEGIN {SelectSort}
    {Copy INPUT to F1 and echo to OUTPUT};
    {Sort F1 to OUTPUT using SelectSort strategy}
  END.  {SelectSort}
```

Design Part 3.1

```
BEGIN {Copy INPUT to F1 and echo to OUTPUT}
  REWRITE(F1);
  WRITE(OUTPUT,'INPUT DATA:');
  READ(INPUT, Ch);
  WHILE Ch <> '#'
  DO
    BEGIN
      WRITE(F1, Ch);
      WRITE(OUTPUT, Ch);
      READ(INPUT, Ch)
    END;
  WRITELN(OUTPUT);
  WRITELN(F1, '#')
END
```

Design Part 3.2

```
BEGIN {Sort F1 to OUTPUT using SelectSort strategy}
  WRITE(OUTPUT, 'SORTED DATA:');
  RESET(F1);
  READ(F1, Ch);
  WHILE Ch <> '#'
  DO {Ch <> '#' and Ch is the first character of F1}
    BEGIN
      {Select the minimum character of F1 for Min,
       and copy the rest of F1 to F2};
      WRITE(OUTPUT, Min);
      {Copy F2 to F1};
      RESET(F1);
      READ(F1, Ch)
    END;
  WRITELN(OUTPUT)
END
```

The DO part status comment (assertion)

```
{Ch <> '#' and Ch is the first character of F1}
```

in Design Part 3.2 will be needed in Design Part 3.2.1, which implements the first task in the DO part BEGIN statement. The first part of the assertion is TRUE because `Ch <> '#'` is the condition for getting into the DO part. The second part of the assertion is TRUE because this point can only be reached from above the WHILE statement, or on its successive iterations; in either case the last two statements executed are

```
RESET(F1);
READ(F1, Ch)
```

so `Ch` must be the first character of `F1`. Without this analysis, the initial assignment to `Min` in 3.2.1 would be suspect.

Design Part 3.2.1

```
BEGIN {Select the minimum character of F1 for Min,
       and copy the rest of F1 to F2}
  REWRITE(F2);
  Min := Ch;
  READ(F1, Ch);
  WHILE Ch <> '#'
  DO
    BEGIN
      {Select minimum(Ch, Min) for Min,
       copy the other character to F2};
      READ(F1, Ch)
    END;
  WRITELN(F2, '#')
END
```

Design Part 3.2.2

```
BEGIN {Copy F2 to F1}
  RESET(F2);
  REWRITE(F1);
  READ(F2, Ch);
  WHILE Ch <> '#'
  DO
    BEGIN
      WRITE(F1, Ch);
      READ(F2, Ch)
    END;
  WRITELN(F1, '#')
END
```

Design Part 3.2.1.1

```
BEGIN {Select minimum(Ch, Min) for Min,
       copy the other character to F2}
  IF Ch < Min
  THEN {Ch is minimum(Ch, Min)}
    BEGIN
      WRITE(F2, Min);
      Min := Ch
    END
  ELSE {Min is minimum(Ch, Min)}
    WRITE(F2, Ch)
END
```

In order to gain confidence in Design Part 3.2, we give four partial execution tables for its behavior (omitting its initial annotating WRITE statement). These tables treat the cases in which F1 has no characters other than #; in which F1 has a single character; and, F1 has two characters, reversed or in order. In each case, Min is assigned the minimum of the first two characters of F1, and the other character is written to F2.

Execution Table for Design Part 3.2, No Characters

After Statement	OUTPUT	F1	F2	Ch	Min
	_	#/	?	?	?
RESET(F1)		#/			
READ(F1,Ch)		#/		#	
WHILE Ch<>'#'					
WRITELN(OUTPUT)	/_				

Execution Table for Design Part 3.2, One Character

After Statement	OUTPUT	F1	F2	Ch	Min
	_	A#/	?	?	?
RESET(F1)		A#/			
READ(F1,Ch)		A#/		A	
WHILE Ch<>'#'					
BEGIN					
BEGIN {3.2.1}					
REWRITE(F2)			_		
Min := Ch					A
READ(F1,Ch)		A#/		#	
WHILE Ch<>#					
WRITELN(F2,'#')			#/_		
END {3.2.1}					
WRITE (OUTPUT, Min)	A_				
BEGIN {3.2.2}					
RESET(F2)			#/		
REWRITE(F1)		_			
READ(F2,Ch)			#/	#	
WHILE Ch<>'#'					
WRITELN(F1,'#')		#/_			
END {3.2.2}					
RESET(F1)		#/			
READ(F1,Ch)		#/		#	
END					
WHILE Ch<>'#'					
WRITELN(OUTPUT)	A/_				

The table illustrates sorting a single-character file by converting a row of values

After Statement	OUTPUT	F1	F2	Ch	Min
RESET(F1)	_	A#/	?	?	?

to the final row

After Statement	OUTPUT	F1	F2	Ch	Min
WRITELN(OUTPUT)	A/_	#/	#/	#	A

Studying the table, it is apparent the single character in F1 is appended to OUTPUT and assigned to Min, regardless of the initial values of F2, Ch, and Min.

Execution Table for Design Part 3.2, Two Characters Reversed

After Statement	OUTPUT	F1	F2	Ch	Min
	_	CA#/_	?	?	?
RESET(F1)		CA#/			
READ(F1,Ch)		CA#/		C	
WHILE Ch<>'#'					
BEGIN					
BEGIN {3.2.1}					
REWRITE(F2)			_		
Min := Ch					C
READ(F1,Ch)		CA#/		A	
WHILE Ch<>'#'					
BEGIN					
BEGIN {3.2.1.1}					
IF Ch < Min					
THEN					
BEGIN					
WRITE(F2,Min)			C_		
Min := Ch					A
END					
END {3.2.1.1}					
READ(F1,Ch)		CA#/		#	
END					
WHILE Ch<>'#'					
WRITELN(F2,'#')		C#/_			
END {3.2.1}					
WRITE(OUTPUT,Min)	A_				
{Copy F2 to F1}		C#/_		#	
* RESET(F1)		C#/			

The details of Design Part 3.2.2 have been omitted in this table. When execution reaches the final row marked by an asterisk, the minimum value from F1 is appended to OUTPUT and the remaining problem has been reduced to sorting a single-character file as in the previous table. Because values of F2, Ch, and Min are immaterial—the single character in F1 (this time C) is appended to OUTPUT and assigned to Min. Therefore, the the two-character table should continue and end with the row:

After Statement	OUTPUT	F1	F2	Ch	Min
WRITELN(OUTPUT)	AC/_	#/	#/	#	C

This is exactly the result obtained by completing the two-character table:

After Statement	OUTPUT	F1	F2	Ch	Min
* RESET(F1)	A_	C#/	C_	#	A
READ(F1,Ch)		C#/		C	
END					
WHILE Ch<>'#'					
BEGIN					
BEGIN {3.2.1}					
REWRITE(F2)			_		
Min := Ch					C
READ(F1,Ch)		C#/		#	
WHILE Ch<>'#'					
WRITELN(F2,'#')			#/_		
END {3.2.1}					
WRITE(OUTPUT,Min)	AC_				
{Copy F2 to F1}		#/_		#	
RESET(F1)		#/			
READ(F1,Ch)		#/		#	
END					
WHILE Ch<>'#'					
WRITELN(OUTPUT)	AC/_				

Therefore, we use the same technique in abbreviating the sorting of the single-character file that arises in the following table.

Execution Table for Design Part 3.2, Two Characters in Order

After Statement	OUTPUT	F1	F2	Ch	Min
	_	AC#/	?	?	?
RESET(F1)		AC#/			
READ(F1,Ch)		AC#/		A	
WHILE Ch<>'#'					
BEGIN					
BEGIN {3.2.1}					
REWRITE(F2)			_		
Min := Ch					A
READ(F1,Ch)		AC#/		C	
WHILE Ch<>'#'					
BEGIN					
BEGIN {3.2.1.1}					
IF Ch<Min					
ELSE					
WRITE(F2,Ch)			C_		
END {3.2.1.1}					
READ(F1,Ch)		AC#/		#	
END					
WHILE Ch<>'#'					
WRITELN(F2,'#')			C#/_		
END {3.2.1}					
WRITE(OUTPUT,Min)	A_				
{Copy F2 to F1}		C#/_			
RESET(F1)	A_	C#/	C#/_	#	A
{Sort one-character file}	AC/_	#/	#/	#	C

The cases shown in the five execution tables above allow the behavior of the design to be inferred for cases that F1 contains more than two characters. The minimum character of F1 is assigned to Min, and the remainder (not necessarily in order) is copied to F2. If there are multiple occurrences of the minimum value in F1, only the first one is assigned to Min. All other characters (including the remaining minimum values) are written to F2.

To support the analysis, SelectSort can be assembled for testing. Perhaps the design is not too long to assemble and test all at once, but combining parts 3, 3.1, 3.2, 3.2.1 seems a better idea. Part 3.2.2 is replaced by a statement that writes the single character # to F1 so that only one iteration occurs.

Development Program 3A

```
PROGRAM SelectSort(INPUT,OUTPUT);
  {Sort the characters preceding the first  #
   from INPUT to OUTPUT.  The program aborts
   if INPUT contains no  #.}
  VAR
    Ch, Min:  CHAR;
    F1, F2:  TEXT;
  BEGIN {SelectSort}
    BEGIN {Copy INPUT to F1 and echo to OUTPUT}
      REWRITE(F1);
      WRITE(OUTPUT,'INPUT DATA:');
      READ(INPUT, Ch);
      WHILE Ch <> '#'
      DO
        BEGIN
          WRITE(F1, Ch);
          WRITE(OUTPUT, Ch);
          READ(INPUT, Ch)
        END;
      WRITELN(OUTPUT);
      WRITELN(F1, '#')
    END;
```

```
  BEGIN {Sort F1 to OUTPUT using SelectSort strategy}
    WRITE(OUTPUT, 'SORTED DATA:');
    RESET(F1);
    READ(F1, Ch);
    WHILE Ch <> '#'
    DO {Ch <> '#' and Ch is the first character of F1}
      BEGIN
        BEGIN {Select the minimum character of F1 for Min,
               and copy the rest of F1 to F2}
          REWRITE(F2);
          Min := Ch;
          READ(F1, Ch);
          WHILE Ch <> '#'
          DO
            BEGIN
              {Select minimum(Ch, Min) for Min,
               copy the other character to F2};
              READ(F1, Ch)
            END;
          WRITELN(F2, '#')
        END;
        WRITE(OUTPUT, Min);
        {For testing write an empty F1...}
        REWRITE(F1) {Test statement};
        WRITELN(F1, '#') {Test statement};
        {...in place of}
        {Copy F2 to F1};
        RESET(F1);
        READ(F1, Ch)
      END;
    WRITELN(OUTPUT)
  END
END.  {SelectSort}
```

Execution

```
INPUT :XBZA#
OUTPUT:INPUT DATA:XBZA#
       SORTED DATA:X
```

The test comes out as expected: the first character of the input is selected and treated as if it were the only one in the file.

When the complete program is assembled, it remains to test the minimum-finding and file-copying portions. If there is trouble in the former, it should be apparent in the (supposedly) sorted output. But since the intermediate files disappear, trouble in the latter would be hidden. Therefore it is wise to insert test statements to make the intermediate files visible:

Development Program 3B

```
PROGRAM SelectSort(INPUT,OUTPUT);
  {Sort the characters preceding the first  #
   from INPUT to OUTPUT.  The program aborts
   if INPUT contains no  #.}
  VAR
    Ch, Min:  CHAR;
    F1, F2:  TEXT;
  BEGIN {SelectSort}
```

```
BEGIN {Copy INPUT to F1 and echo to OUTPUT}
  REWRITE(F1);
  WRITE(OUTPUT,'INPUT DATA:');
  READ(INPUT, Ch);
  WHILE Ch <> '#'
  DO
    BEGIN
      WRITE(F1, Ch);
      WRITE(OUTPUT, Ch);
      READ(INPUT, Ch)
    END;
  WRITELN(OUTPUT);
  WRITELN(F1, '#')
END;
BEGIN {Sort F1 to OUTPUT using SelectSort strategy}
  WRITE(OUTPUT, 'SORTED DATA:');
  RESET(F1);
  READ(F1, Ch);
  WHILE Ch <> '#'
  DO {Ch <> '#' and Ch is the first character of F1}
    BEGIN
      BEGIN {Select the minimum character of F1 for Min,
             and copy the rest of F1 to F2}
        REWRITE(F2);
        Min := Ch;
        READ(F1, Ch);
        WHILE Ch <> '#'
        DO
          BEGIN
            BEGIN {Select minimum(Ch, Min) for Min,
                   copy the other character to F2}
              IF Ch < Min
              THEN {Ch is minimum(Ch, Min)}
                BEGIN
                  WRITE(F2, Min);
                  Min := Ch
                END
              ELSE {Min is minimum(Ch, Min)}
                WRITE(F2, Ch)
            END;
            READ(F1, Ch)
          END;
        WRITELN(F2, '#')
      END;
      WRITE(OUTPUT, Min);
```

```
            BEGIN {Copy F2 to F1}
              RESET(F2);
              WRITE('(Min)   ') {Test of copying};
              REWRITE(F1);
              READ(F2, Ch);
              WHILE Ch <> '#'
              DO
                BEGIN
                  WRITE(Ch) {Test of copying};
                  WRITE(F1, Ch);
                  READ(F2, Ch)
                END;
              WRITELN('#(F1)') {Test of copying};
              WRITELN(F1, '#')
            END;
            RESET(F1);
            READ(F1, Ch)
          END;
        WRITELN(OUTPUT)
      END
    END.  {SelectSort}
```

Execution

```
INPUT :CEDAR#
OUTPUT:INPUT DATA:CEDAR#
       SORTED DATA:A(Min)  EDCR#(F1)
       C(Min)  EDR#(F1)
       D(Min)  ER#(F1)
       E(Min)  R#(F1)
       R(Min)  #(F1)
```

This test shows that the minimum-finding and copying is working as expected. For example, the output line beginning **SORTED DATA** shows that the minimum character found in **CEDAR** is **A**, and the trial minimum **C** that held through **D** and **E** was tossed back in place of the original **A**.

The final `SelectSort` program can now be obtained by deleting the test WRITE statements from Development Program 3B. Wherever possible, testing statements that will be later removed should be inserted so as not to disturb the semicolon or BEGIN statement structure of the program. Then they can be deleted mechanically.

4.2.2 Structural Testing

Testing programs to gain confidence that development and design are proceeding as planned takes considerable thought. It is difficult to know which parts of a design to combine for testing, what special information to print for the test, and above all, how to select test data that will expose mistakes that might have been made. There can be no guarantee that testing will find every difficulty with a program. Everyone who works with programs comes to understand Edsger Dijkstra's statement that tests can only detect the *presence* of errors, not their *absence*. However, the strong point of testing is its more or less mechanical nature, so it is important to have systematic ways to select test data.

Structural testing selects test data based on the program structure. The simplest method is to choose test cases that guarantee that each statement in the program will be executed at least once. The argument for structural testing is a negative one: if statements have *not* been tried, then arbitrary errors could be lurking there, to crop up later. The converse is not true: when all statements *have* been tried, errors can still be lurking there, which the tests did not happen to excite. For all its deficiencies, structural testing is an organized approach to finding test data. If the

programmer has ideas about what might be faulty in a design or development, and can be reassured by testing, those tests are the most useful. When ideas are lacking, it would be foolish not to fall back on structural testing.

For example, consider the test of Development Program 3A with the input **XBZA**. It is evident that each statement was executed; the only parts that could be missed are those inside WHILE statements, should the entry condition prove initially FALSE. The test execution of Program 3B with input **CEDAR#** is a little more difficult to analyze. All WHILE statements enter their DO parts, but within Design Part 3.2.1.1 does the IF statement try both its THEN and ELSE parts? In either case a character is written to `F2`, so this cannot be used to learn which way the condition came out. The THEN part is executed when the trial minimum is replaced, and this happens on the first pass for **CEDAR#**, since **C** is replaced by **A** in the fourth iteration of 3.2.1.1. On the other hand, the ELSE part is executed when the trial minimum is still the real minimum, and this happens on the second iteration, where **C** is less than **E**. Thus **CEDAR#** is an adequate structural test. The analysis also shows that **XHB#** would not be an adequate structural test; the ELSE part would not be covered.

4.2.3 Analysis of SelectSort

In contrast with the families `IFSort` and `MinSort`, whose programs grow larger with the size of the string to be sorted, `SelectSort` will sort a string of any size. That is, `SelectSort`, no larger than `IFSort4` or `MinSort6B`, will sort strings of hundreds or thousands of characters. However, the execution time for `SelectSort` will depend on the length of the input string. The number of steps required can be precisely determined. First, focus on the WHILE statements, which appear in Design Parts 3.1, 3.2, 3.2.1, 3.2.2. Suppressing the other details of the program, the WHILE statements in `SelectSort` can be numbered:

WHILE Statement Tasks in SelectSort

WHILE number	Structure and Task
	`BEGIN`
	`...`
1	`WHILE {Copy INPUT to F1}`
	`...`
2	`WHILE {Repeat so long as F1 isn't empty}`
	`BEGIN`
	`...`
3	`WHILE {Copy all but Min from F1 to F2}`
	`...`
4	`WHILE {Copy F2 to F1}`
	`...`
	`END`
	`...`
	`END`

Suppose there are n input data characters. Then WHILE statement 1 will need n iterations. WHILE statement 2 will likewise need n iterations because `F1` is being shortened by one character in each iteration. The number of iterations of WHILE Statements 3 and 4 depends on the iteration number of WHILE Statement 2. In 2's first iteration, WHILE Statement 3 will need n iterations and WHILE Statement 4 will need n-1 iterations; in the second iteration, WHILE Statement 3 will need n-1 iterations, WHILE Statement 4 n-2 iterations, and so on. Thus, the number of iterations will be as follows:

WHILE statement	Iterations Executed
1	n
2	n
3	n + (n-1) + ... + 1
4	(n-1) + (n-2) + ... + 1 + 0

Next, consider the READ/WRITE statements required in each iteration:

WHILE Statement	READ/WRITEs Executed
1	2
2	1
3	2
4	2

Therefore, the number of READ/WRITE statements N executed by `SelectSort` will be

$$\begin{aligned} N &= 2n + n + 2(n + (n-1) + \ldots + 1) + 2((n-1) + (n-2) + \ldots + 0) \\ &= 5n + 4((n-1) + \ldots + 1) \\ &= 5n + 4n(n-1)/2 \\ &= 5n + 2n^2 - 2n \\ &= 2n^2 + 3n. \end{aligned}$$

Number of READ/WRITE Statements N for input length n

n	N	$2n^2$
5	65	50
10	230	200
20	860	800
100	20,300	20,000
1000	2,003,000	2,000,000

It is clear that the $2n^2$ term controls the value of N as n gets large. We say that N is *on the order of* $2n^2$ for large n. In particular, if each READ/WRITE statement takes the same amount of time, it will take about 100 times as long to sort a 1000-string as it will to do a 100-string; and it will take 10,000 times as long to sort a 1000-string as it will to do a 10-string. If a 10-string can be sorted in a second, the 1000-string will take 10,000 seconds—almost three hours!

Reversing by Selection

`SelectSort` needs only a small modification to become a string-reversing program, which could be called `SelectReverse`. The only modification required is to replace Design Part 3.2.1 (which selects the minimum value from `F1`) by a part with the comment:

```
{Select the last value in F1,
 copying the remainder to F2}
```

It would also probably be appropriate to change the variable holding this value from `Min` to (say) `Last`.

Sorting and Reversing Faster

Either `SelectSort` or `SelectReverse` can be speeded up by a factor of two by using the files `F1` and `F2` symmetrically: that is, by selecting a character out of `F1` while copying the remainder to `F2`, then selecting a character out of `F2` and copying to `F1`, back and forth. An even/odd switch will be required to keep track of which way the copy is being made.

In fact, SelectSort can be speeded up by another arbitrary factor. Internal CHAR variables can be used to hold out not one, but several successive minima on each copying pass. For example, with five variables Min1, Min2, Min3, Min4, Min5, the five smallest values can be retained and printed on each pass. This will reduce the number of READ/WRITE statements by a factor of 5.

Using both these tricks, say with 10 variables and copying back and forth, would reduce the execution time of SelectSort by a factor of 20; from about three hours to ten minutes, for example.

4.2.4 Exercises

4.2.1 Give the contents of OUTPUT and of file F1 on each pass of SelectSort with input:

a) **CLOSET#**

b) **DCBA#**

4.2.2 Design, assemble, and test a program SelectReverse as suggested at the end of Section 4.2.3.

4.2.3 Make up structural test cases for Design Part 2.1.1 of Split (Section 4.1.4), and show why all statements will be executed.

4.2.4 Devise a structural test for IFSort3 (Development Program 2A, Section 3.1.1).

4.2.5 In SelectSort, consider a modification more complex than the one that created SelectReverse. In the first pass, select the first character of the file, in the second pass, the second character of the (modified) file, the third character in the third pass, and so on. If the proper numbered character isn't available (because the modified file is too short), select the last character.

a) What should be the output of such a program on input **123456789** ? Describe the effect this algorithm should have on a general input string.

b) Could a program developed from this algorithm pass the test in (a) and yet be wrong?

c) Could such a program pass a structural test and yet be wrong?

d) How might you *prove* that the algorithm has the general effect you claim in (a)?

4.2.6 Structural testing is not a cure-all for program bugs: a test can be structurally complete and yet show very little about a program. Demonstrate this by writing a program (with no IF or WHILE statements, but it is essential to use READ and WRITE statements) such that (A) *any* test input constitutes a complete structural test; but (B) the program is wrong for every input with a single exception. State what would happen if the single good input were used in a test, then that test pronounced sufficient because it is structurally complete. It is an essential part of this exercise to say what your program is *supposed* to do. Explain why this is crucial.

4.2.7 Design and develop a program that uses three variables Min1, Min2, and Min3 to hold successive smallest characters on each pass of a faster SelectSort, as suggested at the end of Section 4.2.3.

4.2.8 What execution cost (if any) is associated with using more than one variable to hold successive minimum values, as suggested at the end of Section 4.2.3 to cut the number of READ and WRITE statements? For the computer you are using, find out (from documentation or by experiment) how many variable comparisons can be done in the time required for a READ/WRITE. What does this imply about the limits of the several-variable technique for speeding up sorting by selection?

4.2.9 "Shuffled" data result when two sorted files of the same length are merged so that the characters of one file occupy the even positions while those of the other file occupy the odd positions. Design a sorting routine that takes as input a shuffled file and produces as output a sorted file.

4.3 TEXT File Markers

Pascal TEXT files are organized into *lines,* and a means is provided to detect the line boundaries and the end of files. It is the presence of line markers that accounts for the peculiar behavior of READ statements: one extra blank appears to be at the end of each input line. By using the conventional marker # for the end of a file, and putting all data on a single line, it is possible to ignore the real structure of Pascal files. The payment is a loss of power; if the input does not contain #, or has internal line marks, programs could not properly handle it.

4.3.1 End of Line Markers

In execution tables the effect of WRITELN statements on files has been shown with the symbol / to indicate a line break. This symbol is not printed or read from files; it is only a visible way of marking the line boundaries in our tables. We avoided confusion with the actual character / by avoiding statements like

```
WRITELN('/')
```

which would create a line containing the literal character / followed immediately by a line marker. Of course, the actual Pascal machine does not use / for the line marker, but some strange symbol that cannot be confused with any character. This mark is placed into files each time a WRITELN statement is executed. But what happens if an attempt should be made to read the mark itself? Consider the following:

```
REWRITE(F1);
WRITE(F1,'#');
WRITELN(F1);
RESET(F1);
READ(F1,Ch1);
READ(F1,Ch2)
```

Ch1 will have the literal character value #, and Ch2 will have the value of the blank character, the printable replacement for the line marker.

If line boundaries are to be controlled by a program, it is necessary to detect them. It will not do to test for the blank that is read instead of the marker, for a line might contain internal blanks. Pascal provides a standard word EOLN that can be used as a <condition>, to test for an upcoming line marker. When the INPUT cursor is positioned on a marker, so the next READ statement would deliver the substitute blank character, then EOLN has the value TRUE; when the cursor is before this point, so the next READ statement would deliver a character from the input, EOLN has value FALSE. When a TEXT file, say F1, is being read, EOLN(F1) can be similarly used to decide if the cursor is on a line marker.

For example, suppose file F1 contains one line, consisting of the single character **A** (followed of course by the line marker). Here is an execution table showing how EOLN behaves:

After statement	F1	EOLN(F1)	Ch1	Ch2
RESET(F1)	**A**/	FALSE	?	?
READ(F1,Ch1)	**A**/	TRUE	**A**	?
READ(F1,Ch2)	**A**/_	?	**A**	□

where □ is an explicit way of showing the blank character. The question mark for EOLN value after the final line mark has been read means that the value of the <condition> is not defined when no more input remains. Most Pascal machines return something in this case, but there is no agreed-upon value, so EOLN should not be used after the input is completely exhausted.

4.3.2 End of File

Pascal also provides a standard Boolean <condition> `EOF`, whose value tells whether any data remain to be read in a file. `EOF` has value TRUE if the `INPUT` cursor is beyond the last data item (and hence beyond the line marker at the end of the last line); otherwise the value is FALSE. For file `F1`, the <condition> is `EOF(F1)`. For example, if `INPUT` contains one line, data **A**, here is the situation:

After statement	Ch	INPUT	EOF
	?	**A**/	FALSE
READ(Ch)	**A**	**A**/	FALSE
READ(Ch)	□	**A**/_	TRUE

A word of caution is needed about how various Pascal machines treat input lines ending with blank characters. Many machines remove trailing blanks and insert a line character after the last nonblank character. Others change the number of trailing blanks to adjust the line to some internal storage boundary. In the latter case, blanks may appear or disappear from the actual lines.

The syntax for <condition> now becomes:

SR23. <condition> ::= <expression> <comparison> <expression>
¦ `NOT` (<condition>)
¦ (<condition>) `AND` (<condition>)
¦ (<condition>) `OR` (<condition>)
¦ `EOF` ¦ `EOF` (<identifier>) ¦ `EOLN` ¦ `EOLN` (<identifier>)

CR6. A `TEXT` identifier is an <identifier> declared to be of <type> `TEXT`. If a `TEXT` identifier appears in a <READ statement> or a <WRITE statement>, it must be the first <identifier>. A `TEXT` identifier, other than `INPUT` or `OUTPUT`, that appears in a READ statement or WRITE statement must appear in a <declarations>. Only `TEXT` identifiers other than `INPUT` and `OUTPUT` may appear in RESET or REWRITE statements; only `TEXT` identifiers that have appeared in a RESET statement may appear in `EOF` or `EOLN` <condition>s.

4.3.3 Copying Lines

When `INPUT` contains a sequence of lines, the Pascal *READLN statement* may be used to move across the line boundary. The effect of a READLN statement is to skip over all characters through the next line marker, if any, and leave the cursor on the first character of the following line. If there are no line markers left in `INPUT`, a READLN statement will abort. `READLN(F1 ... )` will act on file `F1`.

```
READLN(Ch1, Ch2, Ch3)
```

has the effect of the sequence:

```
READ(Ch1); READ(Ch2); READ(Ch3);
READLN
```

Adding the READLN statement gives:

SR12. <READ statement> ::= `READ` (<identifier list>)
¦ `READLN` (<identifier list>) ¦ `RESET` (<identifier>)

The value of `EOLN` and `EOF` <condition>s, and the actions of READ and READLN statements, are defined only for situations in which tests are made and acted upon *before* the corresponding boundaries pass. Therefore it is essential to use these in the proper pattern, which is given exactly in the program:

```
PROGRAM CopyLines(INPUT,OUTPUT);
  {Copy INPUT to OUTPUT, preserving the line structure.}
  VAR
    Ch: CHAR;
  BEGIN {CopyLines}
    WHILE NOT EOF
    DO
      BEGIN {copy one line}
        WHILE NOT EOLN
          DO
          BEGIN {copy one char}
            READ(Ch);
            WRITE(Ch)
          END;
        READLN;
        WRITELN
      END
  END.  {CopyLines}
```

Execution

```
INPUT :A
       B
OUTPUT:A
       B
```

The outer WHILE statement protects the inner one from a no-data condition using EOF. The inner WHILE protects its DO part's READ statement from the line marker using EOLN. After the inner WHILE is finished with a line, the marker is passed with READLN, and created in the copy with WRITELN. The following execution table shows the details, now showing the line markers in INPUT:

After statement	Ch	INPUT	EOF	EOLN	OUTPUT
	?	**A**/B/	FALSE	FALSE	_
WHILE NOT EOF					
WHILE NOT EOLN					
READ(Ch)	**A**	**A**/B/	FALSE	TRUE	
WRITE(Ch)					**A**_
WHILE NOT EOLN					
READLN		**A**/B/	FALSE	FALSE	
WRITELN					**A**/_
WHILE NOT EOF					
WHILE NOT EOLN					
READ(Ch)	**B**	**A**/B/	FALSE	TRUE	
WRITE(Ch)					**A/B**_
WHILE NOT EOLN					
READLN		**A/B/**_	TRUE	?	
WRITELN					**A/B/**_
WHILE NOT EOF					

The pattern of READ and WRITE statements is different, and more natural in a program that follows the line structure of a file. With the end marker #, the pattern is

```
READ(F1, Ch);
WHILE Ch <> '#'
DO
  BEGIN
    WRITE(F2, Ch);
    READ(F1, Ch)
  END
```

but with the test for end of line, it is:

```
WHILE NOT EOLN(F1)
DO
  BEGIN
    READ(F1, Ch);
    WRITE(F2, Ch)
  END
```

This second pattern more nearly represents the copying operation—read from `F1` then write to `F2`.

4.3.4 BubbleSort

`SelectSort` sorts files of arbitrary length n but executes on the order of $2n^2$ READ/WRITE statements to do so. Is a faster sort possible? The `IFSort` and `MinSort` program families sort files of a fixed length by reading each character once and printing it once. That is, for a fixed n, they take just 2n READ/WRITE statements. Since $2n^2$ is larger than 2n for all $n > 1$, it may be possible to do better that `SelectSort`. A new sorting algorithm will also provide practice in using the pattern of file operations using EOLN.

An attempt to discover a faster sort might begin by noticing that `SelectSort` would still require on the order of $2n^2$ READ/WRITE statements if the input were already sorted, since it would pass over the data in exactly the same way. This observation suggests an initial pass over the data to check if they happen to be sorted. Suppose that the data are all on a single line of file `F1`.

```
BEGIN {Check F1 for sortedness}
  Sorted := 'Y';
  RESET(F1);
  IF NOT EOLN(F1)
  THEN
    BEGIN
      READ(F1, Ch1);
      WHILE NOT EOLN(F1)
      DO
        BEGIN
          READ(F1, Ch2);
          IF Ch2 < Ch1
          THEN
            Sorted := 'N';
          Ch1 := Ch2
        END
    END
END
```

Imagine executing this fragment on a file that is not sorted. At the first place where characters appear out of order, the statement assigning `Sorted` to `N` will be reached. Instead of only flagging it, why not put things right on the spot? `F1` could be copied to `F2` so long as the order is correct, but when `Ch2` should precede `Ch1`, they could be written out reversed. By repeatedly passing over a file, interchanging characters out of order, the file would eventually be sorted. This idea is the basis for an algorithm called "bubble sort," so called because the smaller values seem to bubble up toward the beginning of the file during sorting.

Design Part 4

```
PROGRAM BubbleSort(INPUT,OUTPUT);
  {Sort the first line of INPUT to OUTPUT.}
  VAR
    Sorted, Ch, Ch1, Ch2: CHAR;
    F1, F2: TEXT;
  BEGIN {BubbleSort}
    {Copy INPUT to F1};
    Sorted := 'N';
    WHILE Sorted = 'N'
    DO
      BEGIN
        {copy F1 to F2, checking for sorted and
         reversing adjacent characters out of order};
        {copy F2 to F1}
      END;
    {Copy F1 to OUTPUT}
  END. {BubbleSort}
```

Design Part 4.1

```
BEGIN {copy F1 to F2, checking for sorted and reversing
       the first adjacent characters out of order}
  Sorted := 'Y';
  RESET(F1);
  REWRITE(F2);
  IF NOT EOLN(F1)
  THEN
    BEGIN
      READ(F1, Ch1);
      WHILE NOT EOLN(F1)
      DO {At least two characters remain for Ch1, Ch2}
        BEGIN
          READ(F1, Ch2);
          {Write minimum(Ch1, Ch2) to F2,
                record sorted status}
        END;
      WRITELN(F2, Ch1) {Write last character to F2}
    END
END
```

Design Part 4.1.1

```
{Write minimum(Ch1, Ch2) to F2,
    record sorted status}
IF Ch1 <= Ch2
THEN
  BEGIN
    WRITE(F2, Ch1);
    Ch1 := Ch2
  END
ELSE
  BEGIN
    WRITE(F2, Ch2);
    Sorted := 'N'
  END
```

Design Part 4.2

```
BEGIN {Copy INPUT to F1}
  REWRITE(F1);
  WHILE NOT EOLN
  DO
    BEGIN
      READ(Ch);
      WRITE(F1, Ch);
    END;
  WRITELN(F1)
END
```

The remaining tasks:

Design Part 4.3

`{Copy F1 to OUTPUT}`

Design Part 4.4

`{Copy F2 to F1}`

are similar to Design Part 4.2.

The `BubbleSort` design has an element of mystery to it: two characters are interchanged deep within the program, and the result is that a file is sorted. Will it really work? Of course, the program can be assembled and tested, but the best tests are unreliable for finding subtle design flaws. Perhaps the program works for all but some peculiar pattern of data (say the whole file sorted but the first and last characters interchanged), and that pattern might not be tested. An argument carried out at the design level is the best assurance that a complicated idea works. Think about an input file of length n, and the minimum data item *m* in the file. Unless it is first, *m* must move up one place on each pass and thereby move up to the first place in n passes at most. Similarly, the minimum-but-one item will move up (or down, relative to *m*) to its place in n passes at most; likewise for the others. Therefore, the file will be sorted in at most n^2 passes. (This argument can be improved to show that at worst n passes are needed. See the Exercises.) Once the list is sorted, `Sorted` is set to `Y` and never altered in Design Part 4.1.1.

Is `BubbleSort` faster than `SelectSort`? With the improved argument above, the worst case for `BubbleSort` on a file of length n is n passes and n iterations of the inner WHILE statement for each pass, or a total of n^2 iterations. In the best case, when the input data are already sorted, a total of n iterations is required. Between these extremes, a precise analysis is more difficult. But the number of passes required is at least the maximum distance any character in the input is from its final destination. For a file with characters randomly placed, this maximum distance will probably be a large fraction of n. Therefore, `BubbleSort` can be expected to require a large fraction of n^2 iterations—probably more than `SelectSort` would require. So `BubbleSort` can be faster than `SelectSort` (when the file is close to sorted), but usually it is not.

4.3.5 Exercises

4.3.1 Find out exactly what happens on your Pascal machine for the following:

a) Reading past the line marker with `READ`

b) Testing `EOLN` after the last line marker has passed

c) Reading after `EOF` takes the value TRUE

Be careful of using WHILE statements in these tests, because endless executions are easy to excite.

4.3.2 Write a program that reads multiple lines from `INPUT` and transfers them to `OUTPUT` as a single line, with the literal character `/` marking the line-break positions.

4.3.3 Suppose that a Pascal machine alters the number of characters on actual input lines by adding or removing blanks and the end. Explain why a program designed to copy input lines, like

the one in Section 4.3.3, will not be disturbed. Find out if your Pascal machine does adjust lines in this way.

4.3.4 Assemble and test `BubbleSort` from Section 4.3.4.

4.3.5 Give the contents of file `F1` on each pass of `BubbleSort` with the input data:

a) **CLOSET**

b) **DCBA**

4.3.6 Add a design part to `SelectSort` (Section 4.2.1) that tests at each stage to see if the file is already sorted and if so, avoids further processing. How does this design affect the worst-case behavior? The best-case behavior?

4.3.7 The sorts in this chapter are in ascending order. A descending-order sort is the opposite—the largest character in the file is to appear first, and so on down to the smallest character. Make the design changes to turn

a) `SelectSort` (Section 4.2.1) into a descending sort

b) `BubbleSort` into a descending sort

4.3.8 In Design Part 4.1 of `BubbleSort`, the status comment on the DO part is:

```
{At least two characters remain for Ch1, Ch2}
```

Using appropriate Boolean analysis, give an argument that shows this comment to be accurate.

4.3.9 Improve the analysis of Section 4.3.4 to show that `BubbleSort` makes no more than n passes over a file of length n.

4.3.10 Rewrite `BubbleSort` to handle input files with multiple lines, where

a) Each line is to be sorted separately and printed as a line

b) The whole file is to be sorted together, and printed on a single line.

Do these changes require going back to the design, or are they merely development changes?

4.3.11 Rewrite `SelectSort` so that no # end marker is needed on the input file. Assume that the input is all on one line.

4.4 Chapter Summary

CF (Character and File) Pascal variables may have values that are single characters or files of characters, by declaring them to be <type> `CHAR` or `TEXT`, respectively. A `TEXT` variable provides intermediate storage that is potentially unlimited. However, a rigid discipline is required for manipulating `TEXT` files. They must initially be prepared for writing with a REWRITE statement, closed out with a WRITELN statement, prepared for reading with a RESET statement. The characters retrieved from a file during reading are in exactly the same serial order they were written.

`SelectSort` is a true general-purpose sorting design because it can sort a file of unlimited size, using `TEXT` files for intermediate passes. However, the restricted program families like `MinSort` presented in Chapter 3 are faster: they sort a file of length n using on the order of n operations, while `SelectSort` requires on the order of $2n^2$ operations. A different idea for sorting leads to `BubbleSort`, but its speed is disappointing, sometimes faster than `SelectSort`, but probably not faster on typical data. The execution speed of `SelectSort` can be improved, but its fundamental behavior remains on the order of n^2 operations.

Good program testing requires knowledge of the potential weaknesses in design and development. The best tests are those a programmer uses to try dubious cases, to reassure himself that design ideas work, and that transcription errors have not been made. But it takes familiarity with a design (and a lack of wishful thinking) to devise such ideal tests. When the situation is less than ideal, some systematic form of test selection is helpful, and has at least a chance of catching mistakes. The structural testing method requires that test cases be found to force the execution of every program statement.

Pascal TEXT files can be organized in lines, and file boundaries processed using EOLN to detect line breaks, READLN statements to pass over them, and WRITELN statements to create them. The end of data in a file may be detected using EOF. The pattern of file reads is simpler when these natural boundary markers are used than when an artificial end marker like # is used.

PART II: PROGRAM CALCULUS OF CF PASCAL

The Next Step

Part I described how to write programs to solve elementary problems using a simple programming language, CF Pascal. Part II goes one level deeper by describing how to solve programming problems using mathematical concepts that underlie all programs. Our goal continues to be learning to program in a better way than by trial and error, using a discipline comparable to touch typing. Part I developed the language tools while Part II develops techniques for systematic program design.

The small CF Pascal language was appropriate for Part I because it allowed us to concentrate on the programming and not on the programming language. CF Pascal continues to be appropriate in Part II because it will allow a complete understanding of programs as mathematical objects more complicated than merely character strings. The ideas developed will scale up to programs of any size, in the same way that learning to do long division with three place numbers makes it possible to do any division problem, say with 30-place numbers.

This understanding of programs as mathematical objects is organized into a *program calculus*, a treatment of the semantics of programs rather than their syntax.

What is a Program Calculus?

The introductory calculus course in mathematics is based on the concept of a smooth, real-valued function. The "calculus" involved means calculation, of two related functions, the *derivative* and the *integral* of a suitable given function. These calculations are of critical value to the physical sciences and engineering—indeed, there would be no physical science as we know it today without the real-variable calculus. The derivative function can be interpreted as a rate of change, with profound applications to the description of motion; the integral corresponds to the area under a function's graph, again with far-reaching application. It is nothing short of astonishing that the operations of calculating derivative and integral functions turn out to be inverses.

The *program calculus* also has functions as its fundamental objects, but *discrete* functions, defined not on the real numbers, but on characters and other mathematical objects constructed from characters. The functions are *meanings* for programs. Because any CF Pascal program instructs the Pascal machine to transform input data to output data, the effect of such a program, as a transformation, is a function carrying characters (from `INPUT`) to characters (in `OUTPUT`).

There are two main problems in the program calculus. First, given a program as a syntactic object, find its meaning function. Second, given a meaning function, find a program with that meaning. The second problem leads to systematic techniques for program design, while the first problem leads to systematic techniques for verifying program correctness.

This ability to derive meaning functions from programs in the program calculus is of great value in computer science and engineering. First, it permits a mathematical treatment of program correctness, namely whether a program specifies correct behavior of the computer for every possible input. Debugging incorrect programs is the principal cause of low productivity in the computer industry today. Putting up with incorrect programs is the principal frustration users have with computers today. But the ability to derive functions from programs in a systematic way is even more important. It makes possible a step by step design discipline for writing programs that are correct to begin with, and so do not require debugging. This systematic discipline of design is the principle objective of Part II.

The Mathematical Basis for the Program Calculus

The mathematical basis for the real-variable calculus is the algebra of numbers, and the numerically based coordinate systems of analytic geometry. The functions of the calculus can be pictured geometrically, as curves in Euclidian space.

The mathematical basis for the program calculus is the finite mathematics of discrete structures based on character data. It is mathematics especially suitable for computers and programming. Character data are much richer in human meaning than numbers, and there are no simple pictures to aid in thinking about discrete functions. For example, how do you draw a picture of a sonnet of Shakespeare?

The mathematical basis for the program calculus is just five discrete mathematical structures of character data, namely:

- strings—such as the string of characters in a sentence
- lists—such as the list of words in a sentence (in the order they appear),
- sets—such as the set of words in a sentence (with order and duplication ignored),
- relations—such as the set of pairs of consecutive words in a sentence,
- functions—such as the set of pairs of consecutive words in a sentence in which no word appears twice.

These five structures have the important property that they can be treated at many levels of formality, ranging from pure mathematics to informal English description. Some sets are more easily and precisely described in English than in mathematics, but are sets, no less because of the mode of their description. Many programming problems are better stated in English than mathematics, and we need to be able to treat questions of program correctness and design independent of the mode of description.

The treatment of programs as mathematical objects unifies and simplifies questions of program correctness and design. Computer programs are new to the human experience, but already stand among the most complex logical artifacts mankind has produced. In dealing with complexity, mankind has no other tool equal to mathematics. But just as mathematics has been shaped in the last five hundred years by the needs of the physical sciences, we can be sure that new demands of computer science with have a major impact on mathematics in the next hundred years.

Programs as Mathematical Objects

It is fundamental in computer science to understand programs as mathematical objects that can be reasoned about with mathematical rigor. Mathematically, a program is more than its representation in a programming language such as CF Pascal. The same situations arises with numbers. The number thirteen is a mathematical object which has the property of being thirteen independent of how it might be represented in spoken or written form. For example, the word "thirteen", the digit string 13, or the Roman numeral XIII all represent the number thirteen, but are themselves simply character strings. The difference between thirteen, the number, and "thirteen", the word, is that the number satisfies certain mathematical properties that the word does not, and these properties are independent of the name (the word) we give it. Being the number thirteen means being one more than twelve, having no integral divisors, etc., independent of the representation.

It was an achievement of the first rank in mathematics to isolate the concept of the natural numbers in a set of postulates, as done by the Italian mathematician, Giuseppe Peano, less than a hundred year ago.

Peano's postulates use two undefined terms, *natural number* and *successor,* a literal symbol "0", and variable symbols "x", "y". The postulates are:

z_1) 0 is a natural number.

s_1) 0 is not a successor of any natural number.

s_2) Any natural number has a successor.

s_3) If the successors of any natural numbers x and y are equal then so are natural numbers x and y.

I_1) If a set S of natural numbers contains 0 and if, when S contains any natural number x, S also contains the successor of x, then S contains all the natural numbers.

(The last postulate formalizes the principle of mathematical induction.) Peano then proved from these postulates the familiar properties of order and arithmetic for what we think of as the counting

numbers 0, 1, 2, ... The usual names of the numbers can be thought of as abbreviations for awkward names that occur in Peano's work. Let the successor of natural number n be written s(n). Then the abbreviations are:

1 for s(0)
2 for s(1) = s(s(0))
...
10 for s(9) = s(s(8)) = ...
...
100 for s(99) = s(s(98)) = ...
...

The mathematical properties of 0, 1, 2, ..., 10, ..., 100, ... depend only on 0 and s, and not on their representation in decimal digits. For example, $10 > 8$ because there exists a successor expression for $10 = s(s(8))$; $5 = 3 + 2$ because $5 = s(s(3))$ (s used 2 times). It is remarkable that methods of representing numbers are never mentioned in this development of Peano. The only symbol used is 0, which is quite arbitrary.

Most people deal only with the usual representations of numbers, for example in doing addition, long division, etc. An understanding of the numbers themselves, for example through the Peano postulates, is typically reserved to people who study mathematics in a university. But various procedures such as long division, finding square roots, etc., done with one representation or another, are based on the properties of the numbers themselves. For example, the ordinary grammar school method of finding square roots is derived from an algebraic identity independent of the place notation of the decimal number system. That same identity holds for numbers expressed in Roman numerals, but the Roman numerals do not lead to such convenient calculations as do the Arabic numerals with place notation.

Just as children can learn to find square roots without understanding the difference between a number and its decimal representation, people can learn to write programs without ever knowing about programs as mathematical objects. As long as programs are small and simple, nothing is lost. But when programs become complex, understanding them as mathematical objects, which can be reasoned about with mathematical rigor, provides essential intellectual control.

The property of programs that goes beyond their appearance, beyond their syntax, is their control of computer behavior. A program to sort a string of course has its syntax, but its essence is what it does—sorting. There are many CF Pascal programs that instruct a computer to sort strings in a particular way, just as there are many expressions for the number thirteen (12+1, 8+5, 17-4, etc.). Indeed there are programs in other languages such as BASIC or FORTRAN that define the same ways of sorting strings. Each programming language is a method of representation, just as Roman numerals, decimal digits, etc., represent natural numbers. The functional behavior of a program that sorts strings will be independent of the programming language and the exact form of the program itself. That is, understanding a program as a mathematical object is understanding the functional behavior it induces in a computer.

The Syntax and Semantics of CF Pascal

Since CF Pascal programs are complicated objects, they can be viewed in many ways. A person having trouble getting a program to compile might (at least for the moment) consider the exact syntax of paramount importance. A documentation specialist might be more concerned with the cosmetic arrangement of indentation, and with the style and substance of comments, neither of any concern to the compiler. Someone who had to study and modify a program would pay attention to comments and format, but might also be concerned with its size, because a large program is harder to master than a small one.

Imagining a program in execution brings out an entirely different set of important properties. How long does it run? Could it abort in execution? Which statements actually get executed, and how often? What resources (such as file space or memory) in the actual computer are required to run the program? These execution-related questions depend on the input data used in executing the

program. It usually happens that program behavior is different from input to input, and any set of executions can be misleading if too few inputs have been tried or if the inputs are not representative.

BNF provides an exact way to describe much of the syntax of CF Pascal. Along with context conditions (such as the rule that variables must be declared to be used) BNF syntax rules describe exactly all possibilities for CF Pascal programs. Semantics, the complement to syntax, deals with the meanings of programs. The semantics of the program calculus gives a precise description not of what a program looks like, but rather of what it instructs Pascal machines to do. It is more difficult to describe program-induced actions than it is to describe program form. The idea is simple, however: the description must contain the information from execution tables. It must tell us what happens to variables and files, and which statements make the actions happen. Here we run into the problem of test data: for which inputs is the behavior to be described? The ideal answer is: for every input. A complete semantic description of a program must describe *all* possibilities. This requirement means that an execution table by itself will not do, because a different table is needed for each input, and the number of potential inputs to most programs is unlimited. The person who understands the program could, of course, create any execution table as needed, but what we are after is a way of capturing that understanding formally—in a closed form.

The semantic meaning to be given for a CF Pascal program will be a mathematical relation or function, a set of ordered pairs that defines a correspondence between one collection of data (the inputs) and another collection (the outputs). The meaning of a program is the transformation described by this correspondence: certain outputs are paired with certain inputs because given the input, the program instructs the Pascal machine to compute that output. Thus, the inputs are the potential contents of the file `INPUT`, and the outputs are the resulting contents of `OUTPUT`. The program calculus allows the meaning of a program to be calculated from its form.

CHAPTER 5

THE SEMANTICS OF CF PASCAL

Chapter Preview

Chapter 5 begins the development of the concept of a program's meaning. In order to describe this idea precisely, the mathematical structures of strings, lists, sets, relations, and functions are introduced.

To know the meaning of a CF Pascal program is to be able to predict its output for any given input. A pair of data items (x, y) forms part of the meaning if when INPUT contains data x, the program will execute normally to make OUTPUT contain data y. The collection of all such pairs is the complete *program meaning*. Remember that a program executes normally only if it reaches the final period in execution. If certain INPUT data x_0 leads to an aborted execution, or to an endless execution, then x_0 does not contribute to the meaning; there is no pair (x_0, y) in the meaning, for any y.

Some meanings are easy to express. For example, consider the one-character copy program from Chapter 1:

```
PROGRAM Copy(INPUT,OUTPUT);
  {Write the first character in INPUT to OUTPUT}
  VAR
    Ch: CHAR;
  BEGIN
    READ(Ch);
    WRITELN(Ch)
  END.
```

Its meaning contains all input-output data pairs such that the input string is not empty (has characters or line markers) and the output string contains the first character of the input. If the first character of the input is a line marker, then the output is a line containing a single blank character, so these pairs are in the meaning, too. However, most programs do not have such a convenient description of meaning. They handle an unlimited number of possible inputs, and the rule for the output is complex. (Indeed, the program itself is a rule for getting the output from the input; hence, a large, complex program is likely to have a large, complex meaning.) Some mathematical machinery will be required to deal with programs in general, that of strings, lists, sets, functions, and relations.

5.1 Character Strings

Preview

Character strings are mathematical objects with precise names, operations, and relationships.

New ideas: character string, literal character, literal string, empty string, concatenation, composition, decomposition, sublist.

A *literal string,* or simply *string,* is any sequence of characters whose beginning is marked with † and whose end is marked with †. For example

†**string**†

is a string of six characters, while

† **string** †

is a string of eight characters (beginning and ending with a blank character). It calls attention to a string's length to name it, so these are a 6-string and an 8-string, respectively. The blanks can also be shown explicitly using the symbol □. (In the first part of this book, these two strings would have been shown as **string** and □**string**□ respectively; now we will be more careful and include the markers that identify them as strings.

The string with no characters, the unique 0-string, is called the *empty string,* denoted †† (literally a string with no characters). The string with one blank character † † is not the empty string, of course, but a 1-string.

A *literal character* (or simply *character*) when not in a string will be shown in a special type font. For example, the characters of the 8-string above are in order: □, **s**, **t**, **r**, **i**, **n**, **g**, □. The collection of all possible literal characters must be agreed on in advance and must be finite. It could include symbols from several type fonts, and it could be agreed that hand-written forms for these characters may be used. But the collection of literals may not include the special string marker † ! In this book, the special font will always be used, and when it is necessary to write characters by hand, a good convention is to place a bar over the best facsimile you can make. In hand-written strings overbars are not needed, because the bounding markers imply that the characters within them are drawn from the agreed-on collection.

Characters and strings are not the same thing, but could be confused: a 1-string must be kept distinct from the character it contains. For example, □ is a character, but † † is a 1-string.

It is convenient to give strings names. For example, the definition

$S =$ †**string**†

specifies that string †**string**† has the name S. A given string may have several names, for example

$T =$ †**string**†

along with the above. To avoid confusion in a given discussion, of course the same name should not be used for two different strings. The most important use of names is not to name particular explicit strings, but rather to refer to strings that are fixed but of unknown value. We say, "let X be any string..." when we want to talk about one particular string, but don't care exactly which one it is.

CF Pascal itself provides a mechanism for delimiting strings, using the single quotation mark. The reason for using a different mechanism is that it will be needed to talk about CF Pascal itself. For example,

†**'string'**†

is an 8-string which, used in a WRITE statement, will direct the Pascal machine to print the 6-string

†**string**†.

The 20-string

†**WRITE(Any, 'string')**†

is a <WRITE statement> in which †**WRITE**† is a standard word, †**Any**† is an <identifier>, and †**string**† is data bounded by the 1-string †'†. These statements would be less clear expressed in the Pascal string notation—the last part would be "...data bounded by the 1-string ''''", for example.

CF Pascal provides three operations that can be used to build up strings or extract characters from them:

CF Pascal Statement	String Operation
WRITE ('ABC')	Concatenation
WRITE (Ch)	Composition
READ (Ch)	Decomposition

These string operations form the basis for dealing with character strings as mathematical objects.

5.1.1 String Concatenation

The output data of a program is a character string created by the execution of WRITE statements. A WRITE statement whose <write list> is a <character string> juxtaposes a new string with the current data in OUTPUT. This operation for strings is *concatenation,* denoted &. The concatenation of two strings is the string obtained by juxtaposing the second after the first. For example, given strings

$C =$ †**character**†
$S =$ †**string**†

then

$C \ \& \ S =$ †**characterstring**†
$S \ \& \ C =$ †**stringcharacter**†
$S \ \& \ S =$ †**stringstring**†

By definition, the concatenation of two strings results in a string in every case. One special concatenation is with the empty string. For any string S, both of

†† & S
S & ††

are just S, because †† adds no characters before or after S. That is, for any string S,

†† & $S = S$ & †† $= S$.

Given a string formed by concatenation, it can be used in another concatenation. For example, given the 9-string C and 6-string S above, and

$B =$ † †

(a 1-string of one blank), the 10-string

$C \ \& \ B =$ †**character** †

can be concatenated with S to get

$(C \ \& \ B) \ \& \ S =$ †**character string**†.

Or, first

$B \ \& \ S =$ † **string**†

could be formed, then concatenated with C to get

$C \ \& \ (B \ \& \ S) =$ †**character string**†

which is the same final result. This illustrates a more general fact, that concatenation is *associative*: for any strings X, Y, Z,

$(X \ \& \ Y) \ \& \ Z = X \ \& \ (Y \ \& \ Z)$.

In concatenations the history is lost, and only the final result is visible. When an operation is associative, parentheses are not needed in multiple applications:

$C \ \& \ B \ \& \ S =$ †**character string**†.

Literal strings and their names can be used interchangeably in string expressions, for example,

$C \ \&$ † † & $S = C \ \& \ B \ \&$ †**string**† = †**character string**†.

An operation is *commutative* if its operands can be interchanged without changing the result, but concatenation is not commutative, as

$S \ \& \ C \neq C \ \& \ S$

shows.

5.1.2 Substrings

The string

$W = \dagger$**WRITE(Any, 'string')**$\dagger$

and the string $S = \dagger$**string**$\dagger$ have an obvious relation: the characters of S appear in order within W. That is, S is a *substring* of W. The idea of substring can be captured precisely using concatenation:

> A string p is a substring of string Q if and only if there exist strings X and Y such that $Q = X$ & p & Y.

In the case of S and W above, W is a concatenation of three strings including S:

$W = \dagger$**WRITE(Any, '**$\dagger$ & S & $\dagger$**')**$\dagger$

so that

$X = \dagger$**WRITE(Any, '**$\dagger$
$Y = \dagger$**')**$\dagger$

in the definition, showing that, indeed, S is a substring of W. Or, consider the string

$T = \dagger$**tring**$\dagger$.

Clearly, the definition of substring should single string T out as a substring of S. In this case

$S = \dagger$**s**$\dagger$ & T

so the X of the definition is $\dagger$**s**$\dagger$. But what is Y in the definition? The empty string will do. Since $\dagger\dagger$ adds nothing to a concatenation, we can satisfy the definition of substring by the equation

$S = \dagger$**s**$\dagger$ & T & $\dagger\dagger$

so that X and Y are identified explicitly. The empty string removes the need to define special cases when a substring is found at the beginning or end of a string. The empty string is also a substring of any string, since for any string B,

$B = B$ & $\dagger\dagger$ & $\dagger\dagger$

so $X = B$, $Y = \dagger\dagger$ in the definition above.

5.1.3 String Composition

WRITE statements may perform another operation, attaching the value of a CHAR variable to the string that is the value of OUTPUT. The operation of *composition,* denoted ∇, describes this action. The composition of a string and a character is formed by attaching the character at the end of the string. For example,

$\dagger$**strin**$\dagger$ ∇ **g** = $\dagger$**string**$\dagger$.

Composition provides a way to convert a character to a string, namely to append it to the empty string. For example,

$\dagger\dagger$ ∇ **g** = $\dagger$**g**$\dagger$

This is the general relationship between a character and the 1-string containing that character. Composition with the empty string is so useful that the empty string is suppressed if no confusion arises, making ∇ appear to be a unary, prefix operator:

∇ **g** = $\dagger$**g**$\dagger$
$\dagger$**strin**$\dagger$ & (∇ **g**) = $\dagger$**string**$\dagger$

Composition requires a string as its first operand and a character as its second operand (or in the abbreviated prefix form, a character operand). Thus the following are not defined, for the reasons given:

Not defined	Because
a ∇ **b**	first operand is a character, not a string
∇ †**x**†	operand is a 1-string, not a character
N ∇ ††	both operands are improper

5.1.4 String Decomposition

Just as WRITE statements use string concatenation and composition, READ statements decompose strings. At each execution of a READ statement with a single variable, the available string of characters in INPUT (the characters from the cursor on) is decomposed into two parts. The first character of this string becomes the value of the variable in the READ statement, and the remaining characters of this string become the new string still available to be read. The string operations of *head* and *tail* are precisely the ones used by a READ statement. Head (Θ) and tail (Λ) are unary, prefix operators defined only on *nonempty* strings S:

Θ S is the first character of string S;

Λ S is the substring of S obtained by removing its first character.

For example, given

S = †**string**†
T = †**tring**†

then

Θ S = **s**
Λ S = †**tring**† = T
Θ T = **t**
Λ T = †**ring**†.

Since the tail of a string is a string, it is subject to further decomposition, for example using the above:

Θ(Λ T) = **r**
Λ(Λ T) = †**ing**†
Λ(Λ(Λ T)) = †**ng**†
Θ(Λ(Λ(Λ T))) = **n**.

These examples illustrate that head and tail operations can single out any character of a string. The k^{th} character can be obtained by k-1 tail operations followed by a head operation.

There are two particular conditions to emphasize about head and tail operations:

1: Neither head nor tail operations are defined on the empty string.

2: The result of a head operation is a character, but the result of a tail operation is a string.

Since the head of a string is a character, a 1-string can be formed from it by composition, then concatenated with the tail of the string to get the string back. For example

Θ †**string**† = **s**
Λ †**string**† = †**tring**†

and

∇ (Θ †**string**†) = †**s**†

so

(∇(Θ †**string**†)) & (Λ †**string**†) = †**s**† & †**tring**† = †**string**†.

This example illustrates the fundamental identity for character strings, which involves the operations of concatenation, composition, and decomposition, namely:

For any nonempty string S, $(\nabla(\Theta\ S))\ \&\ (\Lambda\ S) = S$.

The following table summarizes the string operations in examples:

Operation	Symbol	Example
concatenation	&	†**me**† & †**you**† = †**meyou**†
composition	∇	†**plural**† ∇ **s** = †**plurals**†
character to 1-string	∇	∇ **v** = †**v**†
head	Θ	Θ †**Feet**† = **F**
tail	Λ	Λ †**rattle**† = †**attle**†

5.1.5 Exercises

5.1.1 In each part below, name the operation(s) listed and determine the result of the given operations, using E, F, and H as:

E = †**exercises**†
F = †**frequently**†
H = †**help**†

a) $E\ \&\ H$
b) $\Theta\ F$
c) $\Lambda\ H$
d) $\nabla(\Theta\ F)\ \&\ (\Lambda\ H)$
e) $\nabla(\Theta(\Lambda\ E))\ \nabla(\Theta(\Lambda\ F))$
f) $(\Lambda\ H)\ \&\ \nabla(\Theta\ F)\ \&\ \nabla(\Theta(\Lambda\ F))$

5.1.2 Using string operations, create the following strings from the string C = †**computer**†.

a) †**c**†
b) †**omputer**†
c) †**put**†
d) †**come**†
e) †**prom**†
f) †**top**†

5.1.3 Using string operations, isolate the character **b** (not the 1-string †**b**†!) in the string T = †**Tabloid**†.

5.1.4 If $B = X\ \&\ A\ \&\ Y$, then each of X, A, Y should be substrings of B. Explain why this is so, using the formal definition of substring. Hint: $B = ††\ \&\ X\ \&\ (A\ \&\ Y) = (X\ \&\ A)\ \&\ Y\ \&\ ††$.

5.1.5 Are Θ and Λ defined on 1-strings? Give examples and explain.

5.1.6 Show that any string is a substring of itself.

5.1.7 For any string S with more than one character, the expression $\Theta(\Lambda\ S)$ is defined but $\Lambda(\Theta\ S)$ is undefined, a result that depends only on the order in which the operations appear. Write syntax rules in BNF to generate just those string operation sequences in &, ∇, Θ, Λ, and the single variable S, that are defined unless S is too short a string. Write a context condition to handle the latter case. In writing these rules, be careful about which characters and strings are literal, and which part of the BNF notation.

5.1.8 Show that string concatenation can be defined in terms of string composition and string decomposition. Hint: note that if $T = ††$ then $S\ \&\ T = S$, and if $T \neq ††$ find an expression for $S\ \&\ T$ in terms of S, $\Theta\ T$, and $\Lambda\ T$.

5.1.9 Strings with only digit characters **0**, **1**, **2**, ..., **9** can be used to form the conventional representatives for the natural numbers. If D is a string of digit characters, let $val(D)$ be its numerical value. Find a rule for determining $val(D)$ for any D. Hint: if the last character L were removed from D, leaving a leading string C, the the numerical value of D is ten times the numerical value of C with the numerical value of L added. But be careful about applying *val* to characters instead of strings.

5.1.10 Consider a new operator Φ defined by

$$\Phi\ S = \text{the 1-string that begins } S$$

for any nonempty string S. Write an equation that defines Φ in terms of Θ and ∇. Is it so that

$$(\Phi\ S)\ \&\ (\Lambda\ S) = S$$

for any string S? Suppose that $\Phi' = \Phi$ except that $\Phi'(\dagger\dagger) = \dagger\dagger$ (remember that Φ is not defined for $\dagger\dagger$). Why is it false that

$$(\Phi'\ S)\ \&\ (\Lambda\ S) = S\ ?$$

Can you define a new Λ' related to Λ so that the equation above *is* true for Φ' and Λ'? Why would it not be better to get rid of Θ and Λ and use Φ' and Λ' in their place? (Hint: consider $\Theta\ S$.)

5.1.11 Explain the following:

a) Λ †**monkey**† = †**prehensile**†

b) Θ †**state**† = †**president**†

c) †**de**† ∇ †**salt**† = †**sodium**† & †**chlorine**†

d) In (a) to (c), which operations are actually applied to and yield the right kind of objects?

5.2 List Structures

A useful technique in the study of a character string is to consider it a sequence of substrings, each a character string. For example, a Pascal program is made up of words, blanks, and special characters; indeed, as noted in Chapter 1, a Pascal program is unintelligible to humans unless it is so broken up. However, it is essential to preserve the original order so that the substrings might be recombined. An ordered collection of objects is called a *list,* written as a sequence separated by commas and surrounded by angle brackets. For example, given the string

†**this is a list example**†,

the literal list

<†**th**†, †**is** †,†**is** †,†**a** †,†**list example**†>

gives one arbitrary way to cut up the original string. In fact, the Pascal WRITE statement will produce identical output if either the string, or the list of substrings is used, as in:

```
WRITE('this is a list example')
WRITE('th', 'is ', 'is ', 'a ', 'list example')
```

Lists can be given names. For example,

L = <†**th**†, †**is** †, †**is** †, †**a** †, †**list example**†>,
M = <†**this is a list example**†>.

Such list definitions are represented by character strings themselves, including the angle brackets, string markers, equal symbol, and commas, as well as the list name. Symbols and words are used, even in mathematics, in different ways in different contexts—these angle brackets are quite different here than in a discussion about Pascal syntax.

If an object x is in a list K, we denote this fact of *membership* by $x \in K$. If list K has no such object x in any place, we say $x \notin K$. For example, †**is** † $\in L$ but †**is** † $\notin M$, in the examples above. The above list L has five strings, which can be numbered in sequence:

$L_1 =$ †**th**†
$L_2 =$ †**is** †
$L_3 =$ †**is** †
$L_4 =$ †**a** †
$L_5 =$ †**list example**†

while M has one string:

$M_1 =$ †**this is a list example**†

Subscripts will never be used with list names except to mean the members of a list in the positions starting from 1. L is called a 5-list, M a 1-list. A list with no strings, a 0-list, is called the *empty list,* denoted <>. A list with but one string, such as M, is different from that string. That is, M is a 1-list with first (and only) member M_1,

$M = <M_1>$ but $M \neq M_1$.

Note, also that

$<\dagger\dagger> \neq <>$

since $<\dagger\dagger>$ is a 1-list, whose single member string $<\dagger\dagger>_1$ is the empty string ††, but <> is a 0-list and there is no first member $<>_1$.

Although the examples above show lists of strings, list members may also be characters or even other lists. For example, the list,

C = <**c**, **h**, **a**, **r**, **a**, **c**, **t**, **e**, **r**>

is a list of characters that is not the same as the string †**character**†. A list of lists is

$<M, M> =$
<<†**this is a list example**†>, <†**this is a list example**†>>

where the members of this 2-list are the same 1-list used above.

A single list may have members of different kinds; for example, the list

$Q =$ <**A**, <**A**>, †**A**†>

has members that are in order a character, a list, and a string. The list member Q_2 has as first and only member a character:

$(Q_2)_1 =$ **A**.

5.2.1 List Operations

Lists can be manipulated using operations like those defined above for strings: concatenation, composition, and decomposition, for which the same symbols are used.

Concatenation & for lists means juxtaposing elements from two lists to form one. The members of list L & M are the members of L followed by the members of M.

Composition ∇ for lists means attaching an element at the end of a list. The members of list L ∇ x are the members of L followed by x itself. (The special composition with the empty list must be written in full, <> ∇ x, not abbreviated as ∇ x, because it would be confused with the more common character-to-string composition †† ∇ x.)

In decomposition, the *head* operation Θ means the first element, if any, of a list; it is undefined on the empty list. The *tail* operation Λ means a list formed by removing the first element from the original list; it is also undefined on <>.

Sublists are defined using list concatenation as with strings. L is a sublist of M if and only if there exist lists X, Y such that $M = X$ & L & Y.

The following table summarizes the list operations:

Operation	Symbol	Examples
concatenation	&	$\langle \mathbf{A} \rangle \,\&\, \langle \mathbf{A}, \mathbf{B} \rangle = \langle \mathbf{A}, \mathbf{A}, \mathbf{B} \rangle$
		$\langle \dagger \mathbf{12} \dagger, \langle \mathbf{A} \rangle \rangle \,\&\, \langle \mathbf{X} \rangle = \langle \dagger \mathbf{12} \dagger, \langle \mathbf{A} \rangle, \mathbf{X} \rangle$
		$\langle \dagger 1 \dagger, 1 \rangle \,\&\, \langle \rangle = \langle \dagger 1 \dagger, 1 \rangle$
composition	∇	$\langle \mathbf{A} \rangle \nabla \mathbf{B} = \langle \mathbf{A}, \mathbf{B} \rangle$
		$\langle \mathbf{A} \rangle \nabla \langle \rangle = \langle \mathbf{A}, \langle \rangle \rangle$
		$\langle \mathbf{A} \rangle \nabla \dagger\dagger = \langle \mathbf{A}, \dagger\dagger \rangle$
		$\langle \rangle \nabla \mathbf{V} = \langle \mathbf{V} \rangle$
		$\langle \rangle \nabla \langle \dagger\dagger \rangle = \langle \langle \dagger\dagger \rangle \rangle$
head	Θ	$\Theta \langle \mathbf{X}, \langle \mathbf{X} \rangle \rangle = \mathbf{X}$
tail	Λ	$\Lambda \langle \mathbf{X}, \langle \mathbf{X} \rangle \rangle = \langle \langle \mathbf{X} \rangle \rangle$

Although natural numbers are used to denote places in a list, the concept of list does not require natural numbers. The concept of order in a written sequence is simpler and more primitive than natural number. To discover which of two distinct elements k and m precedes the other in a list Q, form the sequence

$$\Theta\, Q,\ \Theta(\Lambda\, Q),\ \Theta(\Lambda(\Lambda\, Q)),\ \ldots$$

and observe whether k or m appears first.

5.2.2 List Descriptions of Files

The mathematical idea of a list is useful in describing Pascal files. Consider `INPUT`. When a program is executing, one essential thing to know about `INPUT` is the portion already read and the portion that remains. In execution tables the division was marked with a cursor.

However, if the contents of `INPUT` is thought of as a 2-list of character strings, it makes the past/future division explicit and allows a precise discussion of what happens when a READ statement is executed. Imagine then that `INPUT` consists of a 2-list L of character strings, first L_1 the part already processed (past string), and second L_2 the part remaining (future string). That is, the contents of `INPUT` is $\langle L_1, L_2 \rangle$. For example, if an execution table shows a single line value for `INPUT` as **ABC**, then

$$L_1 = \dagger \mathbf{A} \dagger,\ L_2 = \dagger \mathbf{BC} \dagger\ \nabla\ /$$

in list and string notation, where the line marker is shown as /.

List operations can be used to give the following description of READ and WRITE statements for one character.

Let the contents of `INPUT` be $L = \langle L_1, L_2 \rangle$ where $L_2 \neq \dagger\dagger$. Then `READ(Ch)` has the following meaning:

(1) `Ch` is given the value $\Theta\, L_2$;

(2) the contents of `INPUT` becomes $\langle L_1\ \nabla(\Theta\, L_2),\ \Lambda\, L_2 \rangle$.

In the example above, following `READ(Ch)`, `Ch` will have value:

$$\Theta\, L_2 = \Theta(\dagger \mathbf{BC} \dagger\ \nabla\ /) = \mathbf{B}$$

and `INPUT` will have value:

$$\begin{aligned} &\langle L_1\ \nabla(\Theta\, L_2),\ \Lambda\, L_2 \rangle \\ &\quad = \langle \dagger \mathbf{A} \dagger\ \nabla(\Theta\ \dagger \mathbf{BC} \dagger\ \nabla\ /),\ \Lambda(\dagger \mathbf{BC} \dagger\ \nabla\ /) \rangle \\ &\quad = \langle \dagger \mathbf{A} \dagger\ \nabla\ \mathbf{B},\ \dagger \mathbf{C} \dagger\ \nabla\ / \rangle \\ &\quad = \langle \dagger \mathbf{AB} \dagger,\ \dagger \mathbf{C} \dagger\ \nabla\ / \rangle \end{aligned}$$

Or, let the contents of `OUTPUT` be $\langle L_1, \dagger\dagger \rangle$, and v be the value of `Ch`. Then `WRITE(Ch)` has the following meaning: the contents of `OUTPUT` becomes $\langle L_1\ \nabla\ v,\ \dagger\dagger \rangle$.

While the 2-list description of a file is precise, it is inadequate to describe the CF Pascal requirements that file operations be used only in proper sequences. For example, READ statements executed against the file `OUTPUT` are illegal. Files declared `TEXT` may be both read and written,

but not in arbitrary patterns. For example, the sequence

```
REWRITE(F1);
WRITE(F1, Ch);
WRITELN(F1);
RESET(F1);
READ(F1, Ch)
```

is legal, but

```
REWRITE(F2);
WRITE(F2, Ch);
READ(F2, Ch)
```

is not, because F2 is not closed out with a RESET statement before the READ.

To capture the file-usage patterns formally requires more information in file descriptions, so we extend the definition to a 3-list consisting of the past string, the future string, and a *file state*, one of the characters **R** or **W** indicating whether the file is **R**eadable or **W**ritable.

The complete rules for file states are given by the transition table below, where a blank entry indicates an illegal operation:

Old file state	Attempted operation and new file state			
	RESET	REWRITE	READ/READLN	WRITE/WRITELN
R	**R**	**W**	**R**	
W	**R**	**W**		**W**

For example, if a file is in state **R**, a REWRITE statement puts it into state **W**. The blank entry under READ/READLN in the **W** row means that a file in state **W** may not be read.

The first operation on any newly declared file must be a REWRITE; before this operation the file is in an unknown state and all other operations are illegal; afterward it is in state **W**. INPUT and OUTPUT are special cases; they are not declared, and always have **R** and **W** states, respectively.

The table below gives the meanings of several WRITE and READ statements using file values in 3-list form. The table does not describe illegal combinations.

Meaning of legal file operations

Statement	Contents of F1		Contents of Ch (v before)
	before	after	
REWRITE(F1)	$\langle x, y, t\rangle$	$\langle \dagger\dagger, \dagger\dagger, \mathbf{W}\rangle$	
WRITE(F1, Ch)	$\langle x, \dagger\dagger, \mathbf{W}\rangle$	$\langle x \nabla v, \dagger\dagger, \mathbf{W}\rangle$	v
WRITELN(F1, Ch)	$\langle x, \dagger\dagger, \mathbf{W}\rangle$	$\langle x \nabla v \nabla /, \dagger\dagger, \mathbf{W}\rangle$	v
RESET(F1)	$\langle x, y, t\rangle$	$\langle \dagger\dagger, x \,\&\, y, \mathbf{R}\rangle$	
READ(F1, Ch)	$\langle x, y \nabla /, \mathbf{R}\rangle$	$\langle x \nabla(\Theta(y \nabla /)), \Lambda(y \nabla /), \mathbf{R}\rangle$	$\Theta(y \nabla \square)$
READLN(F1, Ch)	$\langle x, (y \nabla /) \,\&\, z, \mathbf{R}\rangle$ (y contains no /)	$\langle x \,\&\, (y \nabla /), z, \mathbf{R}\rangle$	$\Theta(y \nabla \square)$

For example, the first line in the table gives the meaning of a REWRITE statement: it transforms a file state of $\langle x, y, t\rangle$ to $\langle \dagger\dagger, \dagger\dagger, \mathbf{W}\rangle$ for any strings x, y, and any file state t.

The execution table below shows successive operations on file F1 in 3-list form:

After statement	F1	Ch
VAR Ch: CHAR;		?
F1: TEXT;	?	
BEGIN		
Ch:='A';		**A**
REWRITE(F1);	<††, ††, **W**>	
WRITE(F1, Ch);	<†**A**†, ††, **W**>	
Ch:='B';		**B**
WRITELN(F1);	<†**A**† ∇ /, ††, **W**>	
RESET(F1);	<††, †**A**† ∇ /, **R**>	
READ(F1, Ch);	<†**A**†, ∇ /, **R**>	**A**
READ(F1, Ch)	<†**A**† ∇ /,††, **R**>	□
END.		

After the first READ statement, EOLN is TRUE. After the final READ statement, EOF is TRUE. Had the program executed one more READ statement, it would have aborted.

5.2.3 Exercises

5.2.1 Given the list of strings L

$L = <$†**this**†, †**is**†, †**a**†, † †, †**very**†, †**simple**†, †**problem**†$>$

use list and string operations to create the strings for each part below.

a) †**this is a simple problem**†

b) †**this is a messy problem**†

5.2.2 Let N be the 5-list:

$N = <$**1**, **2**, **3**, **4**, **5**$>$.

What is $\Theta(\Lambda(\Lambda(\Lambda\ N)))$? Why is it nonsense to write

$\Theta\ N + \Theta\ N = 2$?

5.2.3 For a list or string X it makes sense to write

$\Lambda(\Lambda(...\Lambda\ X...))$

for any number of applications of Λ. However, this expression is not always defined. Explain the circumstances under which it isn't. Why doesn't it similarly make sense to write

$\Theta(\Theta(...\ \Theta\ X\ ...))$?

Write an expression using nested Θ that does make sense and captures the same kind of meaning as the repeated Λ expression. State clearly what your expression means.

5.2.4 Suppose you want to select the third character in a string X. Write an expression that does so, using the string operators. What is the value of your expression when $X =$ †**TW**†?

5.2.5 Write an expression whose value is the substring of any string X that is the initial 3-string of X. What is the value if $X =$ †**12**†?

5.2.6 Convert the following INPUT file values from execution-table form into 3-list form:

a) **ABC□E**

b) **ABC_DE**

c) **□BCDE**

d) **ABCDE_**

e) _ (not □!)

5.2.7 Convert the following OUTPUT file values from execution-table form into 3-list form:

a) **ABC**_

b) _ (not ␣!)

5.2.8 Given that the contents of F1 is <†**ABC**†, †**CD**†, **W**>, and of Ch is **E**, determine their contents, if defined, after each of the following:

a) REWRITE(F1)

b) RESET(F1)

c) WRITE(F1,Ch)

d) WRITELN(F1)

e) READ(F1,Ch)

5.2.9 Give a formalization of INPUT as a 4-list $<x, y, z, w>$, where x and w are as in the 3-list formalization, y is the future string without its first character, and z is the first character of the future string. Describe the action of a single-character READ statement in your formalization.

This example of two different formalizations of the same idea should raise the important question: can one formalism be "right" and another "wrong," or are all equally valid? Which formalism do you prefer in this case?

5.3 Set Structures

Preview

Sets are collections of objects with order and duplication disregarded. Sets are basic mathematical objects which are used in constructing relations and functions.

New ideas: set, set membership, subset, power set, set operations, union, intersection, difference.

Strings and lists are collections of objects in which order and duplication are essential. A collection in which order and duplication plays no role, where objects are simply grouped together, none first, none repeated, is a *set*. The members of a set are written between braces {}. Of course, they must be set down in some order within the braces, but that order is insignificant. For example,

{**A**, **B**} and {**B**, **A**}

are two ways of writing the same set of characters.

Any list defines a set whose members are the list elements. For example,

$L =$ <†**th**†, †**is** †, †**is** †, †**a** †, †**list example**†>

has the set of members

$S =$ {†**th**†, †**is** †, †**a** †, †**list example**†}.

in which the duplicate list member †**is** † appears only once in the set. Since it is immaterial in what order the strings appear in the set notation,

$S =$ {†**list example**†, †**a** †, †**is** †, †**th**†}
$=$ {†**th**†, †**a** †, †**list example**†, †**is** †}

among others. The size of a set may be indicated in talking about it; here S is a 4-set.

Set members are not necessarily all the same kind of objects. For example, the 3-set

{**A**, †**mixed**†, <†**can**†, {†**of**†, †**beans**†}>}

has a character member, a string member, and a 2-list member. The 2-list has in order a string member and a 2-set member, the latter with string members. (Now that sets have been introduced, they are allowed to be members of lists, too.)

The fundamental relationship between an object and a set is *membership,* for which we use the symbol $\in$. In the example above,

$\dagger$**list example**$\dagger \in S$,

but

$\dagger$**list**$\dagger \notin S$.

The set with no members at all, written {}, is called the *empty set.* The empty set {} is different from the empty list <>. For example, <> can be concatenated with a list, but {} cannot be. A 1-set is called a *singleton.* A singleton is different from its one member, and is different from the list with the that same member. For example,

$\dagger$**string**$\dagger \neq \{\dagger$**string**$\dagger\} \neq <\dagger$**string**$\dagger>$.

A set S is a *subset* of a set T, denoted $S \subseteq T$, if every member of S is a member of T. That is, $S \subseteq T$ if and only if for every x, $x \in T$ if $x \in S$.

Sets S and T are equal, denoted $S = T$, if they have exactly the same members. That is, $S = T$ if and only if $S \subseteq T$ and $T \subseteq S$.

The empty set is a subset of every set, including itself, since it has no members that can fail to be members of the other set. That is, $\{\} \subseteq S$ for all S.

If sets S and T have no common members, then they are said to be *disjoint.*

5.3.1 Rules for Describing Sets

An alternative to listing the members of a set explicitly is the so-called *set builder* notation, in which we place inside the braces a name for a generic member of the set, and a condition describing this member, separated by a colon. For example,

$T = \{x$: x is any word in the string $\dagger$**this is a rule**$\dagger\}$
$= \{\dagger$**this**$\dagger, \dagger$**a**$\dagger, \dagger$**is**$\dagger, \dagger$**rule**$\dagger\}$.

This is read, "the set of all x such that..." To be in the set, a member must satisfy the condition that follows the colon.

The condition describing set membership may have several parts. For example, the set

$U = \{x$: $x = <y, z>$, y is any word in the string $\dagger$**this one list**$\dagger$, z is any character in the string $\dagger$**AB**$\dagger\}$

means that x must simultaneously satisfy three conditions:

1. x must be a 2-list
2. x_1 must be a word in $\dagger$**this one list**$\dagger$
3. x_2 must be a character in $\dagger$**AB**$\dagger$

As a result, since $\dagger$**this one list**$\dagger$ has three distinct words and $\dagger$**AB**$\dagger$ has 2 distinct characters, set U will contain 6 (3×2) members, enumerated:

$U = \{<\dagger$**this**$\dagger$, **A**>, <$\dagger$**this**$\dagger$, **B**>, <$\dagger$**one**$\dagger$, **A**>, <$\dagger$**one**$\dagger$, **B**>, <$\dagger$**list**$\dagger$, **A**>, <$\dagger$**list**$\dagger$, **B**>$\}$.

A set's generic member can take the form of a structure such as a list or a set. Part of the condition may thus be given in the generic member. For example, the definition of U could also be

$U = \{<y, z>$: y is any word in the string $\dagger$**this one list**$\dagger$, z is any character in the string $\dagger$**AB**$\dagger\}$.

When more information can be given in the generic member's form, and thus eliminate conditions that follow the colon, it is usually done. For example,

$\{<x, y>: x \text{ is a string}, y = x \nabla /\}$

would usually be written

$\{<x, x \nabla />: x \text{ is a string}\}$.

One set with sets as members occurs so frequently that it is given a name. The *power set* of any set S is the set of all subsets of S, denoted power(S). For example, if

$T = \{\dagger\texttt{this}\dagger, \dagger\texttt{a}\dagger, \dagger\texttt{is}\dagger, \dagger\texttt{rule}\dagger\}$

then

$$\text{power}(T) = \{U : U \subseteq T\}$$
$$= \{\ \{\}, \{\dagger\texttt{this}\dagger\}, \{\dagger\texttt{is}\dagger\}, \ldots, \{\dagger\texttt{a}\dagger, \dagger\texttt{is}\dagger, \dagger\texttt{rule}\dagger\}, T\}$$

contains (since T is a 4-set) one 0-set (the empty set), four 1-sets (4 ways to form a 1-set from 4 members), six 2-sets, four 3-sets, and one 4-set (T itself), or 1+4+6+4+1 = 16 members. It is no accident that 16 is 2^4, hence the name power set. For each of the four members of T, there are two choices in forming a subset, to include the member or exclude it. Therefore the number of members in the power set of T will be $2 \times 2 \times 2 \times 2 = 16$. In general if S is an n-set, power(S) will be a 2^n-set.

The set-builder notation is most useful when the sets to be described are large, or infinite. For example,

$\{p: p \text{ is a CF Pascal program}\}$

cannot be described by listing its members.

5.3.2 Set Operations

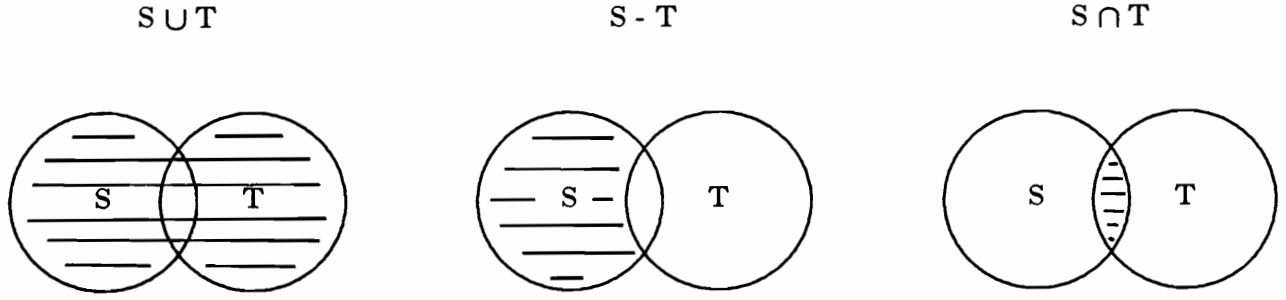
Sets can be combined to make new sets, and three operations are especially useful:

Union ($\cup$): $S \cup T = \{x: x \in S \text{ or } x \in T\}$

Difference (-): $S - T = \{x: x \in S, x \notin T\}$

Intersection ($\cap$): $S \cap T = \{x: x \in S, x \in T\}$

These operations are easily visualized in *Venn diagrams,* in which the component sets are shown as circles, overlapping with the set resulting from the operation shaded:



If two sets lie inside the same third set, then their union, difference, and intersection also remain inside that set: if $S \subseteq U$ and $T \subseteq U$, then

$S \cup T \subseteq U$,
$S - T \subseteq U$,
$S \cap T \subseteq U$.

For any set S, the empty set behaves as follows:

$S \cup \{\} = \{\} \cup S = S$
$S - \{\} = S, \{\} - S = \{\}$
$S \cap \{\} = \{\} \cap S = \{\}$

Union and intersection are commutative, that is:

$S \cup T = T \cup S,$
$S \cap T = T \cap S,$

and associative, that is:

$R \cup (S \cup T) = (R \cup S) \cup T,$
$R \cap (S \cap T) = (R \cap S) \cap T,$

so parentheses are not required to specify the order of multiple unions or intersections.

Difference is neither commutative nor associative.

As examples of these facts, let

$S = \{$†**this**†, †**is**†, †**a**†, †**set**†$\}$
$T = \{$†**this**†, †**is**†, †**another**†, †**set**†$\}$.

Then

$S \cup T = T \cup S = \{$†**this**†, †**is**†, †**a**†, †**set**†, †**another**†$\}$
$S - T = \{$†**a**†$\}$
$T - S = \{$†**another**†$\}$
$S \cap T = \{$†**this**†, †**is**†, †**set**†$\}$
$(S - T) \cup (S \cap T) = \{$†**this**†, †**is**†, †**a**†, †**set**†$\}$
$(S - T) \cap (S \cap T) = \{\}$

The following table summarizes the characteristics of set operations:

Operator	Symbol	Definition	Properties
Union	$\cup$	$S \cup T = \{x: x \in S \text{ or } x \in T\}$	associative commutative
Intersection	$\cap$	$S \cap T = \{x: x \in S, x \in T\}$	associative commutative
Difference	-	$S - T = \{x: x \in S, x \notin T\}$	neither

5.3.3 Relations and Functions

Any set of 2-lists (or pairs) is called a *relation,* and relations will be especially useful in discussing the meaning of programs.

The English word "relation" can mean a rule of comparison, such as "equality," "is a subset of," etc. Formal relations, that is, sets of 2-lists, can describe these informal rules precisely, by including exactly those pairs whose members do stand in the intended relationship to each other. For example the relationship between characters and 1-strings containing those characters is described by the relation

$C = \{<x, \nabla\, x>: x \text{ is a character}\} = \{<$**a**, †**a**†$>, <$**b**, †**b**†$>, ...\}$.

Since a relation is a set, an empty relation $\{\}$ is a possibility, for example, the relation that pairs odd natural numbers with their squares that happen to be even (there are none!). Furthermore, set operations apply to relations; if r, s are relations then so are

$r \cap s, r \cup s, r - s,$

because they are just sets of ordered pairs, too.

Relations with the special property that each first member is paired with a *unique* second member, are called *functions*. Formally, a relation r is a function if and only if wherever $\langle x, y\rangle \in r$ and $\langle x, z\rangle \in r$, then $y = z$. In this case, each first member can serve as a name for the second member in the context of the relation. For example, the relation C between characters and 1-strings defined above is a function.

The set operations also apply to functions. Although the result of set operations on the sets of ordered pairs that are functions will necessarily be another set of ordered pairs, and hence a relation, the result need not be a function. If f, g are functions, $f \cap g$ and f - g must also be functions, but $f \cup g$ may or may not be a function. For example, define the head relation

$$H = \{\langle \Theta\, y,\ y\rangle\text{: } y \text{ is a string}\} = \{\langle \mathbf{a}, \dagger\mathbf{a}\dagger\rangle, \langle \mathbf{a}, \dagger\mathbf{aa}\dagger\rangle, \ldots\}$$

and let C be the relation above. Then:

$$C \cap H = C$$

is a function,

$$H - C = \{\langle \Theta\, y,\ y\rangle\text{: y is a string with at least 2 characters}\}$$

is a relation that is not a function;

$$C - H = \{\}$$

is the empty function, and

$$C \cup H = H$$

is a relation (these specific results are deducible from the fact that $C \subseteq H$).

The set of first members of a relation (or function) is called its *domain*; the set of second members is called its *range*. In the members of a relation, say $\langle x, y\rangle \in r$, x is called an *argument* of r and y is called a *value* of r.

When $\langle x, y\rangle \in r$ and y is the only value for argument x, the *value notation* is:

$$y = r(x).$$

This is read "y is the value of r for argument x," or more briefly, "y is the value of r at x."

Given an arbitrary relation r and an arbitrary argument x, three possibilities are:

1. $x \notin \text{domain}(r)$. In this case r is *undefined* at x.
2. $x \in \text{domain}(r)$ and there exist distinct y, z such that $\langle x, y\rangle \in r$ and $\langle x, z\rangle \in r$. In this case r is *multiply defined* at x.
3. $x \in \text{domain}(r)$ and there is a unique pair $\langle x, y\rangle \in r$. In this case r is *uniquely defined* at x, and $y = r(x)$.

Thus a function is a relation that is uniquely defined for all members of its domain.

Three kinds of special functions are important.

The *empty function* $\{\}$ has no arguments or values; that is

$$\text{domain}(\{\}) = \{\},\ \text{range}(\{\}) = \{\}.$$

An *identity function* is a function I such that if $x \in \text{domain}(I)$ then $I(x) = x$.

A *constant function* is a function whose range is a 1-set, that is, every argument has the same value.

Since they are sets, relations and functions can be described by enumerations or by rules. For example,

$$r = \{\langle \dagger\mathbf{ball}\dagger, \dagger\mathbf{bat}\dagger\rangle, \langle \dagger\mathbf{ball}\dagger, \dagger\mathbf{game}\dagger\rangle, \langle \dagger\mathbf{game}\dagger, \dagger\mathbf{ball}\dagger\rangle\}$$

is a relation (because all its members are 2-lists) with

$$\text{domain}(r) = \{\dagger\mathbf{ball}\dagger, \dagger\mathbf{game}\dagger\}$$
$$\text{range}(r) = \{\dagger\mathbf{ball}\dagger, \dagger\mathbf{game}\dagger, \dagger\mathbf{bat}\dagger\}$$

However, r is not a function because two distinct value strings are paired with the argument

†**ball**†. An example of a relation defined by a rule is:

$s = \{<x, y>$: word x immediately precedes word y in the string
†**this is a relation that is not a function**†$\}$

In this case the relation can also be enumerated:

$s = \{<$†**this**†, †**is**†$>, <$†**is**†, †**a**†$>, <$†**a**†, †**relation**†$>, <$†**relation**†, †**that**†$>,$
$<$†**that**†, †**is**†$>, <$†**is**†, †**not**†$>, <$†**not**†, †**a**†$>, <$†**a**†, †**function**†$>\}$.

It is evident that the string used in the definition of s is a true statement about s. The following rule defines a function:

$f = \{<x,y>$: the first instance of word x immediately precedes word y
in the string †**this is a relation that is also a function**†$\}$

which can be enumerated:

$f = \{<$†**this**†, †**is**†$>, <$†**is**†, †**a**†$>, <$†**a**†, †**relation**†$>,$
$<$†**relation**†, †**that**†$>, <$†**that**†, †**is**†$>, <$†**also**†, †**a**†$>\}$.

5.3.4 Exercises

5.3.1 Given

$A = \{x$: x is any four-letter word in the string †**this is a rule**†$\}$

a) Give A explicitly.

b) Is †**is**† $\in A$?

c) Is †**this**† $\in A$?

5.3.2 Given

$S = \{$†**this**†, †**is**†, †**a**†, †**set**†$\}$
$T = \{$†**this**†, †**is**†, †**another**†, †**set**†$\}$

determine the value of the expression in each part:

a) T - S

b) $(S - T) \cap (S \cup T)$

c) $(S \cap T) - (T - S)$

5.3.3 Give an example to show that set difference (-) is not associative.

5.3.4 Show that the following conditions statements about sets S and T are equivalent (that is, all are true or all are false):

a) $S \subseteq T$

b) $S \cap T = S$

c) $S \cup T = T$

d) $S - T = \{\}$

Which of these translates into the simplest English description of the relationship between S and T? Give the translation.

5.3.5 Describe the circumstances under which each of the following equations holds for arbitrary sets S and T:

a) $S \cap T = T$

b) $S - T = T$

c) $S \cup T = S \cap T$

d) $S - T = T - S$

5.3.6 Explain why domain and range are themselves functions and give their domains and ranges.

5.3.7 Consider the problem of obtaining from a relation R, a function F that reflects a maximal part of R, in the sense that $F \subseteq R$ and if $<x, y> \in R$, there is a z such that $<x, z> \in F$. Explain why each of (a) and (b) fails to capture this idea:

a) $F = \{<x, y>: <x, y>$ is the first pair in R with first element $x\}$

b) $F = \{<x, y>: <x, y> \in R$ and for each $<x, z> \in R$, the length of y is less than the length of $z\}$

c) Give a definition that does capture this idea.

5.3.8 Given any list M, define

$S^M = \{x: x$ occurs in $M\}$.

a) Give an example of an n-list M in which S^M has n elements.

b) Give an example of an n-list M in which S^M has n-2 elements.

c) Why can't it be the other way around in b)?

5.4 Program Meanings

Preview

Relations and functions are vital to describing the meaning of a program. Using these concepts, a notation is devised to describe the meaning of a program. For simple programs, the meaning will be clear, but from these simple examples, a full theory will emerge.

New ideas: box notation, program *vs.* meaning of program.

The set of input-output pairs from all possible normal executions of a Pascal program is called the *program meaning,* or as the case may be the *program function* or *program relation.* It is important to distinguish the program meaning from its members. On a particular input, the Pascal machine instructed by a Pascal program may deliver an output. But the program meaning is much more than a way of expressing the outcome of one particular execution; it expresses *all possible* executions of the Pascal machine under its program's rule.

A program can accept input broken into lines and produce output also broken into lines. Thus the pairs in the program meaning will be pairs of lists of character strings.

5.4.1 The Box Notation

Any CF Pascal program is a character string presented to the Pascal machine. For example,

$P =$

†**PROGRAM PrintHello(INPUT,OUTPUT);BEGIN WRITELN('HELLO')END.**†

is the string for one of the first programs of Chapter 1. CF Pascal programs have a special form (each begins with the string †**PROGRAM**† and ends with the string †**END.**†, for example), so there will be no confusion in abbreviating a program by omitting the string markers. The special font used for programs is a further reminder that a program is really a special kind of string. Thus the program above will also be written:

$P =$

```
PROGRAM PrintHello(INPUT,OUTPUT);BEGIN WRITELN('HELLO')END.
```

or even:

```
P = PROGRAM PrintHello(INPUT,OUTPUT);
      BEGIN
        WRITELN('HELLO')
      END.
```

The string P itself represents the *syntax* of the program; its *meaning* will be written $\boxed{P}$. The meaning $\boxed{P}$ is a set of 2-lists (ordered pairs) of character-string lists, in which the arguments represent program inputs, and the values represent program outputs. That is,

$\boxed{P}$ = {<L, M>: for input list of strings L, P executes normally to yield output list of strings M}.

This *box notation* for the program's meaning keeps the syntax and semantics together, but clearly distinguishes one from the other. For `PrintHello` above,

$\boxed{P}$ = {<L, M>: L is any list of strings and M is the 1-list <†**HELLO**†>}
= {<L, <†**HELLO**†>>: L is any list of strings}.

Putting the box around the program display directly:

$\boxed{P}$ = `PROGRAM PrintHello ... END.`.

Using the value notation, since $\boxed{P}$ is a function,

`PROGRAM PrintHello ... END.`(L) = <†**HELLO**†>

for any list of strings L.

The box notation hides the way a program instructs the Pascal machine on how to execute, and displays only what the execution accomplishes. The term "black box" is often used to describe a mechanism seen only on the outside, in terms of its inputs and outputs. Thus this notation is appropriate for indicating the input-output meaning of a program. For example, the program R:

```
PROGRAM PrintHelloInSteps(INPUT,OUTPUT);
  BEGIN
    WRITE('HE');
    WRITE('L');
    WRITELN('LO')
  END.
```

has the same meaning as $\boxed{P}$, that is, $\boxed{R} = \boxed{P}$. From the outcome of any execution whatsoever, it is impossible to tell whether P or R was executed.

The program named R above also has a CF Pascal name `PrintHelloInSteps`. But since the string †**PrintHelloInSteps**† appears as part of the string R, it is better not to use `PrintHelloInSteps` as the program name with the box notation.

5.4.2 Determining Program Meanings Directly

Subsequent chapters will develop a program calculus which allows the calculation of $\boxed{P}$ for any program P. However, it is often possible in simple cases (or as part of a complicated case) to determine $\boxed{P}$ informally, to capture precisely what a program P does from an informal analysis.

Any program using a READ statement may encounter line markers or the end of file, which complicates the program's meaning. For example, let Q be the program:

```
PROGRAM ReadWriteRead(INPUT,OUTPUT);
  VAR
    Ch: CHAR;
  BEGIN
    READ(Ch);
    WRITELN(Ch);
    READ(Ch)
  END.
```

The input file may be organized into a list of strings (lines) each terminated by a line marker. There are several cases to consider. If the list is empty, the first READ statement aborts. If INPUT is a 1-list and its one string is the empty string, the first READ statement reads the line marker (as a blank) and the second READ statement aborts. If the first string of INPUT is not empty, then the first READ statement gets the first character and the second READ statement gets either the second character or the line marker if there is no second character; in this case Q executes normally and writes the first character of the first string of INPUT to OUTPUT. If INPUT is a 2- or more list and the first string is empty, then the first READ statement gets the line marker for the first string and the second READ statement gets the line marker; in this case Q executes normally and writes a blank to OUTPUT. This analysis is summarized in the following set notation for $\boxed{Q}$ as a mapping from INPUT to OUTPUT:

$$\boxed{Q} = \{<L, M>: L \neq <>, L_1 \neq \dagger\dagger, M = <\nabla(\ominus L_1)>\} \cup$$
$$\{<L, M>: L \neq <>, \Lambda\, L \neq <>, L_1 = \dagger\dagger, M = <\dagger\square\dagger>\}.$$

(Recall that L_1 is the first component of list L, here the first line of INPUT.)

The domain and range of Q can be determined directly from $\boxed{Q}$ by finding the first and second members of its pairs. The domain is a list of strings:

$$\text{domain}(\boxed{Q}) = \{L: L \neq <>, L_1 \neq \dagger\dagger\} \cup \{L: L \neq <>, \Lambda\, L \neq <>, L_1 = \dagger\dagger\}$$

For range($\boxed{Q}$), the second members are 1-lists of 1-strings, and the one character is either the initial character of an input string, or is blank. Thus

$$\text{range}(\boxed{Q}) = \{M: M \text{ is a 1-list containing a 1-string}\}.$$

Notice how the program meaning simply omits those pairs for which there is no normal execution, in the example, those for which L is the empty list or a 1-list with an empty first string.

As a further example, consider R:

```
PROGRAM CopyTwice(INPUT,OUTPUT);
  VAR
    Ch: CHAR;
  BEGIN
    READ(Ch);
    READ(Ch);
    WRITE(Ch);
    WRITELN(Ch)
  END.
```

To write the meaning of R, find the second character (if any) of the list of strings L in INPUT, taking account of line markers.

Conditions	Comments
$L = <>$	abort
$L \neq <>$	examine L_1
$L_1 = \dagger\dagger$	examine existence of L_2
$\Lambda\ L = <>$	abort
$\Lambda\ L \neq <>$	examine L_2
$L_2 = \dagger\dagger$	print blank twice
$L_2 \neq \dagger\dagger$	print $\Theta\ L_2$ twice
$L_1 \neq \dagger\dagger$	examine L_1
$\Lambda\ L_1 = \dagger\dagger$	print blank twice
$\Lambda\ L_1 \neq \dagger\dagger$	print $\Theta(\Lambda\ L_1)$ twice

Each comment draws a conclusion or points to a next condition. As a result of this analysis, the meaning of R is

$$\boxed{R} = \{<L, M>: L \neq <>, L_1 = \dagger\dagger, \Lambda\ L \neq <>, L_2 = \dagger\dagger, M = <\dagger\square\square\dagger>\} \cup$$
$$\{<L, M>: L \neq <>, L_1 = \dagger\dagger, \Lambda\ L \neq <>, L_2 \neq \dagger\dagger, M = <(\nabla(\Theta\ L_2))\ \nabla\ (\Theta\ L_2)>\} \cup$$
$$\{<L, M>: L \neq <>, L_1 \neq \dagger\dagger, \Lambda\ L_1 = \dagger\dagger, M = <\dagger\square\square\dagger>\} \cup$$
$$\{<L, M>: L \neq <>, L_1 \neq \dagger\dagger, \Lambda\ L_1 \neq \dagger\dagger, M = <(\nabla(\Theta(\Lambda\ L_1)))\ \nabla\ (\Theta(\Lambda\ L_1))>\}.$$

5.4.3 Exercises

5.4.1 Determine the program meaning in each part. Provide a description in English and mathematically. (Although these programs were written before we described multiple-line input in CF Pascal, include all possible cases in the meaning.)

a) `VERTICALGREETING` (Section 1.2).

b) `CopyChar` (Section 1.3)

c) `Copy2` (Section 1.3).

d) `CopyReversed` (Section 1.3)

5.4.2 Determine the program meaning for the program of Reading Problem 3 (Section 1.6.1) using as much formal precision as you can.

5.4.3 Suppose that a program P contains *no* WRITE or WRITELN statements. What can you say about $\boxed{P}$? (Hint: there is a complication in domain($\boxed{P}$) should P not execute successfully.)

5.4.4 Suppose a program P contains *no* READ or READLN statements. What can you say about its execution? Can this be said in terms of $\boxed{P}$?

5.4.5

a) Write a program P such that $\boxed{P}$ is the constant function, $\boxed{P}(x) = <\dagger\dagger>$ for all x.

b) Write a program Q such that $\boxed{Q}$ is undefined on all inputs.

c) Write a program R such that $\boxed{R}(<\dagger\mathbf{A}\dagger>) = <\dagger\dagger>$, but $\boxed{R}$ is undefined for all other inputs.

5.5 Chapter Summary

Strings, lists, and sets provide a vocabulary for precisely defining the meanings of CF Pascal programs. Strings and lists share the operations of concatenation (&), composition (∇), and decomposition (Θ and Λ). By considering `TEXT` files as 3-lists, precise definitions can be given for operations that manipulate files, including all difficult cases of line boundaries and legal sequences of operations.

Sets have operations union ($\cup$), intersection ($\cap$), and difference (-). Sets are basic mathematical objects used in constructing relations and functions that are used to give the meanings of programs.

The box notation distinguishes the syntax aspect of a program P (its character string) from the semantic aspect $\boxed{P}$ (its meaning function).

CHAPTER 6

THE CALCULUS OF SEQUENTIAL PROGRAMS

Chapter Preview

Chapter 6 begins the development of the program calculus for sequential programs and culminates in design rules for creating BEGIN statements that execute correctly.

Although it is possible to determine the program meaning of a Pascal program directly from the definition for small programs, larger and more complex programs are difficult to analyze directly. The next chapters are devoted to a systematic method for determining program meanings a step at a time. The parts of programs, declarations, procedures, and statements can be analyzed in isolation, then combined to form the meaning of the whole. This treatment begins with the decomposition of programs into blocks, declarations, and BEGIN statements, then continues with statements of increasing complexity, namely sequential statements, conditional statements, and iteration statements.

A program containing no IF or WHILE statements is a *sequential program,* which has execution tables whose rows never change. That is, since the control flow necessarily passes straight through the program, the program parts entered in each row are always the same, no matter what input data is supplied. Values in the columns of the tables for one program, however, may differ depending on the input data, for each column represents a value history of a program variable. Each row of an execution table corresponds to a program part, which transforms one execution state (in which the variables have the values of the previous row) to another (the values in the current row). These transformations are the functions or relations that provide the basis for the program calculus.

6.1 The Meanings of Program Parts

Preview

Simple programs have simple meanings, but dealing with complex programs requires decomposing them. The process for determining the meaning of a complex program is to find the meanings of its constituent parts and then combine these meanings.

New ideas: execution states, composition of relations and functions, inverse functions, transpose, the meaning of blocks and declarations.

The goal of the program calculus is to calculate $\boxed{P}$ for any CF Pascal program P in terms of the components of P. CF Pascal syntax provides the ideal method of breaking down a program into appropriate parts whose meanings can be found, then combined. The intermediate transformations, each corresponding to a single fragment of Pascal program, are called *part meanings,* to distinguish them from the program meaning. These part meanings will be symbolized by boxing portions of programs.

6.1.1 Execution States

In an execution table, variables other than `INPUT` and `OUTPUT` may be present to describe the actions of the Pascal machine as declarations are encountered, assignments executed, etc. The strings attached to `INPUT` and `OUTPUT` in the table are part of the program's working data, but not all of it. For example, consider the program defined in part as:

```
PROGRAM Copy1(INPUT,OUTPUT);
  VAR
    Ch1: CHAR;
    Ch2: CHAR;
    F1: TEXT;
  BEGIN
    REWRITE(F1);
    Ch1 := 'B';
    READ(Ch2);
    WRITE(F1,Ch2);
    ...
  END.
```

Its execution table for input <ABC> begins:

Partial Execution Table

After execution of	INPUT	OUTPUT	F1	Ch1	Ch2
PROGRAM Copy1(...);	**ABC/**	_			
VAR					
Ch1: CHAR;				?	
Ch2: CHAR;					?
F1: TEXT;			?		
BEGIN					
REWRITE(F1);			_		
Ch1 := 'B';				**B**	
READ(Ch2);	**ABC/**				**A**
WRITE(F1,Ch2);			**A**_		
...					
END.					

The values of the variables INPUT, OUTPUT, F1, Ch1, Ch2 shown in the table are an essential part of the history of execution.

The essential information in an execution table is a listing of the current variables and their values—the information in one table row. An appropriate formal object to use in describing a row of variables and values is a set of identifier-value pairs. For example, the data state following the READ statement in the table above can be represented as a 5-set of pairs:

$$s = \{\langle\dagger\mathbf{INPUT}\dagger, \langle\dagger\mathbf{A}\dagger,\dagger\mathbf{BC}\dagger\ \nabla\ /,\mathbf{R}\rangle\rangle, \langle\dagger\mathbf{OUTPUT}\dagger, \langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle\rangle,$$
$$\langle\dagger\mathbf{F1}\dagger, \langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle\rangle, \langle\dagger\mathbf{Ch1}\dagger,\ \mathbf{B}\rangle, \langle\dagger\mathbf{Ch2}\dagger,\ \mathbf{A}\rangle\}$$

Each identifier is paired with an appropriate value. For example, INPUT is paired with a 3-list having a past string of †**A**†, a future string of †**BC**† ∇ /, and a file state of **R**. This set of pairs derived from an execution table is called an *execution state*. Execution state s is a function, whose domain is a set of identifiers, and whose range is a set of values, of the kinds appropriate to the identifiers.

The execution state changes from line to line in an execution table. The central problem of the program calculus is to calculate execution states from line to line, using the program parts that define them. The meaning of a program part is the function or relation that relates the execution states between two lines of the execution table. For example, the execution state following s, after the WRITE statement, is

$$t = \{\langle\dagger\mathbf{INPUT}\dagger, \langle\dagger\mathbf{A}\dagger,\dagger\mathbf{BC}\dagger\ \nabla\ /,\mathbf{R}\rangle\rangle, \langle\dagger\mathbf{OUTPUT}\dagger, \langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle\rangle,$$
$$\langle\dagger\mathbf{F1}\dagger, \langle\dagger\mathbf{A}\dagger,\dagger\dagger,\mathbf{W}\rangle\rangle, \langle\dagger\mathbf{Ch1}\dagger,\ \mathbf{B}\rangle, \langle\dagger\mathbf{Ch2}\dagger,\ \mathbf{A}\rangle\}$$

That is, t is the same as s except for the pair $\langle\dagger\mathbf{F1}\dagger, \langle\dagger\mathbf{A}\dagger,\dagger\dagger,\mathbf{W}\rangle\rangle$, which reflects the action of the WRITE statement. The program calculus will provide a WRITE-statement rule for deriving t from s without reference to any other part of the program leading up to s.

Since the execution states s and t are functions, the value notation can be used. For example:

$s(\dagger\texttt{INPUT}\dagger) = \langle\dagger\texttt{A}\dagger,\dagger\texttt{BC}\dagger\ \nabla\ /,\texttt{R}\rangle$

$t(\dagger\texttt{Ch1}\dagger) = \texttt{B}$

The execution states for the early lines of this execution table are less obvious. In the program header line, `F1, Ch1, Ch2`, have no values because they have not yet been declared. Therefore, the execution state for this line makes no mention of `F1, Ch1, Ch2`, but is

$$u = \{\langle\dagger\texttt{INPUT}\dagger, \langle\dagger\dagger,\dagger\texttt{ABC}\dagger\ \nabla\ /,\texttt{R}\rangle\rangle, \langle\dagger\texttt{OUTPUT}\dagger, \langle\dagger\dagger,\dagger\dagger,\texttt{W}\rangle\rangle\}$$

a function whose domain is $\{\dagger\texttt{INPUT}\dagger, \dagger\texttt{OUTPUT}\dagger\}$. Now, since u must eventually be transformed into s, it is evident that the meaning of a program part may change not only the values of functions with identical domains but may also change the domains of functions.

The declarations of `CHAR` variables `Ch1` and `Ch2` and file `F1` in the execution table above show the symbol ? as value. This symbol means that although these identifiers now have values, the values are unknown. That is, the execution state following the declaration of `Ch1` is one from a whole collection of states. Each possibility has the same form:

$$v = \{\langle\dagger\texttt{INPUT}\dagger, \langle\dagger\dagger,\dagger\texttt{ABC}\dagger\ \nabla\ /,\texttt{R}\rangle\rangle, \langle\dagger\texttt{OUTPUT}\dagger, \langle\dagger\dagger,\dagger\dagger,\texttt{W}\rangle\rangle, \langle\dagger\texttt{Ch1}\dagger, x\rangle\},$$

where x is a character. One of these states, for some particular value of x, results from the declaration of `Ch1`, but which one is not known. It still makes sense to use the value notation for some identifiers, for example

$$v(\dagger\texttt{OUTPUT}\dagger) = \langle\dagger\dagger,\dagger\dagger,\texttt{W}\rangle,$$

but for `Ch1`, the best we can say is that $v(\dagger\texttt{Ch1}\dagger)$ is some unknown character value.

It will be convenient to use the symbol ? for the unknown initial value of a variable, and write

$$v = \{\langle\dagger\texttt{INPUT}\dagger, \langle\dagger\dagger,\dagger\texttt{ABC}\dagger\ \nabla\ /,\texttt{R}\rangle\rangle, \langle\dagger\texttt{OUTPUT}\dagger, \langle\dagger\dagger,\dagger\dagger,\texttt{W}\rangle\rangle, \langle\dagger\texttt{Ch1}\dagger, ?\rangle\}$$

as an abbreviation. But it is important to remember that ? is not a literal character value.

Many of the special symbols used to write execution states as sets of lists can be eliminated without loss of precision; the resulting states are easier to read. The following abbreviations will be used:

1. The angle brackets will be omitted around identifier-value pairs; instead the identifier and value will be separated by a dot (·).
2. The identifiers will be shown in the special program font, not as strings.
3. String bounding markers will be omitted when a value cannot be confused with a character.

Using these abbreviations, the state s following the READ statement above, instead of being shown as

$$s = \{\langle\dagger\texttt{INPUT}\dagger, \langle\dagger\texttt{A}\dagger,\dagger\texttt{BC}\dagger\ \nabla\ /,\texttt{R}\rangle\rangle, \langle\dagger\texttt{OUTPUT}\dagger, \langle\dagger\dagger,\dagger\dagger,\texttt{W}\rangle\rangle,$$
$$\langle\dagger\texttt{F1}\dagger, \langle\dagger\dagger,\dagger\dagger,\texttt{W}\rangle\rangle, \langle\dagger\texttt{Ch1}\dagger,\ \texttt{B}\rangle, \langle\dagger\texttt{Ch2}\dagger,\ \texttt{A}\rangle\}$$

will be abbreviated

$$s = \{\texttt{INPUT}\cdot\langle\dagger\texttt{A}\dagger,\texttt{BC}/,\texttt{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\texttt{W}\rangle,\ \texttt{F1}\cdot\langle\dagger\dagger,\dagger\dagger,\texttt{W}\rangle,\ \texttt{Ch1}\cdot\texttt{B},\ \texttt{Ch2}\cdot\texttt{A}\}.$$

When line markers are not of particular interest, they will be shown as part of the strings they end.

6.1.2 The Meaning of the Program Header and Period

The first and last parts of a program (the program header and the final period) have a special role to play in the program calculus. They match the interior part meanings, whose domains and ranges are execution states, to the external program meaning, whose domain and range are lists of strings. The correspondence between internal and external values makes use of `INPUT` and `OUTPUT`, which occur in the execution states as variables, but whose contents outside the program are lists of strings.

Whatever list of strings L is applied as external program input is converted by the program header into the future string of `INPUT` with line markers appropriately inserted. That is, a list

$$L = \langle L_1, L_2, \ldots, L_n \rangle$$

of input strings is converted into the internal string

$$(L_1 \nabla /) \,\&\, (L_2 \nabla /) \,\&\, \ldots \,\&\, (L_n \nabla /)$$

which becomes the future-string value for INPUT.

Given program header $H =$ †**PROGRAM** † & N & †**(INPUT,OUTPUT)**† for any identifier named N, the part meaning of H is:

$$[\![H]\!] = \{\langle\langle L_1, L_2, \ldots, L_n\rangle, s\rangle : s = \{\text{INPUT}\cdot\langle\dagger\dagger, x, \mathbf{R}\rangle,\ \text{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle\} \text{ where } x = (L_1 \nabla /) \,\&\, (L_2 \nabla /) \,\&\, \ldots \,\&\, (L_n \nabla /)\}.$$

When the string markers are omitted, it is usual to write:

$$H = \text{PROGRAM } N \text{ (INPUT, OUTPUT)}$$

and say, "where N is an identifier." This is an abbreviation for the form that shows each part of the string explicitly, and for the phrase "N names a string that conforms to the syntax of <identifier>." For example,

$$[\![\text{PROGRAM Copy1 (INPUT, OUTPUT)}]\!](\langle\dagger\mathbf{AB}\dagger,\dagger\mathbf{CD}\dagger\rangle) = \{\text{INPUT}\cdot\langle\dagger\dagger,(\mathbf{AB} \nabla /) \,\&\, (\mathbf{CD} \nabla /),\mathbf{R}\rangle,\ \text{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle\}.$$

At the completion of execution, the final period converts the past string of OUTPUT into program output, namely a list of character strings where the line markers define the separation of members.

Thus the meaning of the final period is defined to be:

$$[\![.]\!] = \{\langle s, \langle L_1, L_2, \ldots, L_n\rangle\rangle : s(\text{OUTPUT}) = \langle (L_1 \nabla /) \,\&\, (L_2 \nabla /) \,\&\, \ldots \,\&\, (L_n \nabla /), \dagger\dagger, \mathbf{W}\rangle\}.$$

Thus the pairs in the state s other than the one with argument OUTPUT are simply lost. For example,

$$[\![.]\!](\{\text{INPUT}\cdot\langle\mathbf{ABC} \nabla /,\dagger\dagger,\mathbf{R}\rangle,\ \text{OUTPUT}\cdot\langle(\mathbf{AB} \nabla /) \,\&\, (\mathbf{CD} \nabla /),\dagger\dagger,\mathbf{W}\rangle,\ \text{Ch1}\cdot\mathbf{B}\}) = \langle\dagger\mathbf{AB}\dagger, \dagger\mathbf{CD}\dagger\rangle,$$

so the printed result will be:

AB
CD

In summary, the domains and ranges for the part meanings introduced so far are given in the following table:

Table of Domains and Ranges of Part Meanings

Program part	*Domain*	*Range*
program	lists of strings	lists of strings
program header	lists of strings	execution states
final period	execution states	lists of strings

6.1.3 Compositions of Relations and Functions

The part meaning of a program header converts an external list of strings to a value attached to INPUT in an execution state; the part meaning of the final period converts a value attached to OUTPUT in an execution state to an external list of strings. In between, the remainder of the program transforms the first execution state into the final one. That is, the header function supplies the input for the main part of the program, whose output in turn is the input for the period part function. The result of relations and/or functions acting one after the other in this way is called their *composition*.

The composition of relations r and s, denoted $r \circ s$, is a new relation of all possible pairs $\langle x, z\rangle$ such that for some intermediate value y, $\langle x, y\rangle \in r$ and $\langle y, z\rangle \in s$. Informally, in set-builder notation,

$$r \circ s = \{\langle x, z\rangle : \text{for some } y, \langle x, y\rangle \in r, \langle y, z\rangle \in s\}.$$

The new idea of "for some y" can be defined in terms of an idea already presented, the domain of a relation. Consider the relation between pairs and intermediate values:

$$t = \{\langle\langle x, z\rangle, y\rangle : \langle x, y\rangle \in r, \langle y, z\rangle \in s\}.$$

An argument $\langle x, z\rangle$ is paired with y in t just when the intermediate y does exist, and thus the meaning of $r \circ s$ is all possible arguments of t,

$$r \circ s = \text{domain}(t).$$

The program calculus is being defined to capture the intuitive meaning in execution tables, but there is a profound difference between the box functions of the calculus and the tables. The box functions describe program meaning *in its entirety,* while the execution table gives only one instance of the meaning (for a particular input value). The information in the table is obtained by evaluating the box function for a single argument. Starting with this argument, the part meanings produce each intermediate set of values in the table. Thus the whole table represents the composition of the part meanings, but *evaluated for one argument.* The composition of box functions themselves is a function, not merely the value of a function, and as such it can be used in calculating the meaning of larger program parts, whereas a value such as the table provides is useless in such calculations.

The definition of composition uses an intermediate value, which can be eliminated for functions. If r is uniquely defined at x, the definition can be rewritten as follows, using value notation:

$$r \circ s = \{\langle x, z\rangle : \text{for some } y,\ y = r(x), \langle y, z\rangle \in s\}$$

and if in addition s is uniquely defined at y,

$$r \circ s = \{\langle x, z\rangle : \text{for some } y,\ y = r(x),\ z = s(y)\}.$$

Substituting for y,

$$r \circ s = \{\langle x, s(r(x))\rangle : r \text{ uniquely defined at } x,\ s \text{ uniquely defined at } r(x)\}$$

and in this form the intermediate y does not appear. When r and s are uniquely defined in this way, the resulting values are also unique. For functions, all values are uniquely defined, and thus the composition of two functions is a function.

The value notation provides a way to find the value of a composition of functions. To calculate

$$(f \circ g)(x)$$

first find $y = f(x)$, then find $g(y)$.

For example, the function defined by the composition

$$\boxed{\texttt{PROGRAM Any(INPUT,OUTPUT)}} \circ \boxed{.}$$

can be calculated directly for any list of strings L. First find

$$y = \boxed{\texttt{PROGRAM Any(INPUT,OUTPUT)}}(L),$$

and then $\boxed{.}(y)$. That is,

$$\boxed{\texttt{PROGRAM Any(INPUT,OUTPUT)}} \circ \boxed{.}(L) = \boxed{.}(y)$$

where $y = \boxed{\texttt{PROGRAM Any(INPUT,OUTPUT)}}(L)$.

The result is

$$y = \{\texttt{INPUT}\cdot\langle\dagger\dagger, u, \texttt{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger, \dagger\dagger, \texttt{W}\rangle\}, \text{ and } \boxed{.}(y) = \langle\dagger\dagger\rangle$$

independent of the string u created from the list L. The past string of the OUTPUT value is empty, and that is enough for the result. Thus this composition is a constant function with value $\langle\dagger\dagger\rangle$ for any argument list of strings L.

Composition is associative. That is, for any relations r, s, t

$$(r \circ s) \circ t = r \circ (s \circ t)$$

because

$$\begin{aligned}(r \circ s) \circ t &= \{<x,u>: \text{for some } z, <x,z> \in (r \circ s), <z,u> \in t\} \\ &= \{<x,u>: \text{for some } z, \text{for some } y, <x,y> \in r, <y,z> \in s, <z,u> \in t\} \\ &= \{<x,u>: \text{for some } y, <x,y> \in r, \text{for some } z, <y,z> \in s, <z,u> \in t\} \\ &= \{<x,u>: \text{for some } y, <x,y> \in r, <y,u> \in (s \circ t)\} \\ &= r \circ (s \circ t).\end{aligned}$$

Therefore, compositions of more than two relations need not be parenthesized, and $r \circ s \circ t$ can be calculated from either $(r \circ s) \circ t$ or $r \circ (s \circ t)$, whichever may be more convenient.

Composition is not commutative. However, an identity function and the empty function commute with any relation. That is for any relation s:

if domain(s) $\subseteq$ domain(I), then $s \circ \mathrm{I} = \mathrm{I} \circ s = s$

$s \circ \{\} = \{\} \circ s = \{\}$.

Functions or relations whose actions exactly undo one another's in composition are called *inverses*. That is, s is a right inverse of r if and only if

$$r \circ s = \mathrm{I}_r,$$

where I_r is the identity function over domain(r).

The *transpose* of a function or relation r, denoted r^{T}, is r with all its pairs reversed, that is,

$$r^{\mathrm{T}} = \{<x, y>: <y, x> \in r\}$$

The transpose may be an inverse, and it is in this role that it will be used in the program calculus. A useful test is the following:

If the transpose of a relation r is itself a function, then that transpose is a right inverse for r.

To see this, suppose the transpose of r is a function. Then by the definition of composition,

$$\begin{aligned}r \circ r^{\mathrm{T}} &= \{<x, z>: \text{for some } y, <x, y> \in r, <y, z> \in r^{\mathrm{T}}\} \\ &= \{<x, z>: \text{for some } y, <y, x> \in r^{\mathrm{T}}, <y, z> \in r^{\mathrm{T}}\}\end{aligned}$$

Now, since r^{T} is a function, any two values paired with the same argument in it must be identical, as are x and z paired with y above. Thus

$$\begin{aligned}r \circ r^{\mathrm{T}} &= \{<x, x>: \text{for some } y, <x, y> \in r, <y, x> \in r^{\mathrm{T}}\} \\ &= \{<x, x>: \text{for some } y, <x, y> \in r\} \\ &= \mathrm{I}_r, \text{ the identity function over domain}(r).\end{aligned}$$

Therefore r^{T} is a right inverse for r.

6.1.4 The Meaning of Declarations

The syntactic object that falls between a program header and the final period is a CF Pascal block. The part meaning of a block therefore transforms the initial execution state into the final one. This transformation may have several distinct parts, resulting from variable and procedure declarations and the BEGIN statement of which the block is composed.

The effect of a declaration on the execution state is to add to that state one or more new identifiers. Before a declaration occurs, the identifiers have no existence; after the declaration they may have values. The part meaning of a declaration must therefore alter the domain of the execution state, adding an identifier. It will be convenient to separately define the meaning of VAR declarations for CHAR, VAR declarations for TEXT, and PROCEDURE declarations.

The meaning of VAR declarations will be defined using relations that are not functions. The relation will pair all possible values with the identifier being declared, capturing the idea that the value is unknown. For a CHAR variable N, define:

$$\boxed{\texttt{VAR } N \texttt{ : CHAR}} = \{<s, t>: t = s \cup \{<N, x>\} \text{ for some character } x\}.$$

(Remember that this is an abbreviation for: the string N conforms to the syntax of <identifier> and the string

$$V = \dagger\textbf{VAR}\dagger \;\&\; N \;\&\; \dagger\textbf{: CHAR}\dagger$$

is being considered.) In the notation using ? for the value of N:

$$\boxed{\texttt{VAR } N \texttt{ : CHAR}} = \{<s, t>: t = s \cup \{N\cdot?\}\},$$

it is not so obvious that every possible character is paired with the identifier. For example, let s be an argument of VAR Ch: CHAR. Then s is paired with values

$s \cup \{\texttt{Ch}\cdot\textbf{A}\}$,
$s \cup \{\texttt{Ch}\cdot\textbf{B}\}$,
...
for all possible characters.

Although $\boxed{\texttt{VAR } N \texttt{ : CHAR}}$ is not a function, $\boxed{\texttt{VAR } N \texttt{ : CHAR}}^{\mathrm{T}}$ *is* a function, because its pairs are of the form

$<s \cup \{N\cdot\textbf{A}\}, s>$
$<s \cup \{N\cdot\textbf{B}\}, s>$
...

and each argument *has* a unique value.

The treatment of a file declaration is similar. Let N be an identifier. Define:

$$\boxed{\texttt{VAR } N \texttt{ : TEXT}} = \{<s, t>: t = s \cup \{N\cdot<x,y,z>\} \text{ for some strings } x \text{ and } y, \text{ and some } z \in \{\textbf{R},\textbf{W}\}\}.$$

The abbreviated notation attaches the 3-list <?,?,?> to the newly declared identifier.

Declarations of more than a single variable are a straightforward generalization: several identifiers are added to the state, each paired with all possible characters or 3-lists, as appropriate. This can be expressed by composition. For example,

$$\boxed{\texttt{VAR Ch1, Ch2: CHAR}} = \boxed{\texttt{VAR Ch1: CHAR}} \circ \boxed{\texttt{VAR Ch2: CHAR}}$$
$$= \{<s, t>: t = s \cup \{\texttt{Ch1}\cdot?\} \cup \{\texttt{Ch2}\cdot?\}\}$$

which includes pairs such as

$$<s, s \cup \{\texttt{Ch1}\cdot\textbf{C}\} \cup \{\texttt{Ch2}\cdot\textbf{F}\}>$$

and all other combinations of two character values.

Suppose that

$$x = \{\texttt{INPUT}\cdot<\dagger\dagger,\dagger\dagger,\textbf{R}>,\ \texttt{OUTPUT}\cdot<\dagger\dagger,\dagger\dagger,\textbf{W}>\}$$

is an argument of the relation VAR Ch: CHAR. Then x is paired in this relation with values

$$\{\texttt{INPUT}\cdot<\dagger\dagger,\dagger\dagger,\textbf{R}>,\ \texttt{OUTPUT}\cdot<\dagger\dagger,\dagger\dagger,\textbf{W}>\} \cup \{\texttt{Ch}\cdot c\}$$

for all characters c. We abbreviate this information with the formula:

$$\boxed{\texttt{VAR Ch: CHAR}}(\{\texttt{INPUT}\cdot<\dagger\dagger,\dagger\dagger,\textbf{R}>,\ \texttt{OUTPUT}\cdot<\dagger\dagger,\dagger\dagger,\textbf{W}>\})$$
$$= \{\texttt{INPUT}\cdot<\dagger\dagger,\dagger\dagger,\textbf{R}>,\ \texttt{OUTPUT}\cdot<\dagger\dagger,\dagger\dagger,\textbf{W}>,\ \texttt{Ch}\cdot?\}.$$

Procedure declarations apparently do not affect the values displayed in an execution table. When later the procedure identifier is used in a procedure statement, however, the <body> of the procedure must be invoked. It is the business of the declaration to record the information that makes this later action possible, namely the text of the <body>. This text is attached to the

procedure identifier as value in the execution state. For procedure identifier N and declared <body> T, define:

$$\boxed{\texttt{PROCEDURE } N \texttt{ ; } T} = \{<s, t>: t = s \cup \{<N, T>\}\}.$$

In the definitions of declaration part meaning, and throughout the further definitions to follow, the programs and program parts must be syntactically correct. No meanings are assigned to strings that do not obey the syntax and context rules. By assigning meanings only to correct programs, we avoid repeating all the complications of syntax in semantics. For example, if it were possible to encounter a declaration of an already-declared variable, the definition would have to do something special about this situation. But since redeclaration is a syntax error, the semantic definitions can ignore it.

6.1.5 The Meaning of Blocks

A block may contain zero or more variable and/or procedure declarations, but it must conclude with a single BEGIN statement. The simplest <block> is one that contains neither declarations nor statements:

```
BEGIN
END
```

In an execution table, this block would not alter any variable values, so as a transformation from execution states to execution states its meaning is an identity function I:

$$\boxed{\texttt{BEGIN END}} = I.$$

When a block contains declarations, identifier-value pairs are added to the execution state before the BEGIN statement is executed, then removed after its execution. For example, variables are added by the part meaning $\boxed{\texttt{VAR ...}}$ defined in the previous section; then later, $\boxed{\texttt{VAR ...}}^{\mathrm{T}}$ is just what's needed to delete those variables.

A simple block with declarations is:

```
VAR
  Ch: CHAR;
BEGIN
END
```

with the part meaning:

$$\boxed{\texttt{VAR Ch: CHAR; BEGIN END}}$$
$$= \boxed{\texttt{VAR Ch: CHAR}} \circ \boxed{\texttt{BEGIN END}} \circ \boxed{\texttt{VAR Ch: CHAR}}^{\mathrm{T}}$$

namely, a composition that first adds Ch·? to the state, then performs the identity transformation, then removes Ch and its value from the state. Beginning with execution state

$$\{\texttt{INPUT}\cdot<\dagger\dagger, \nabla\,/, \mathbf{R}>,\ \texttt{OUTPUT}\cdot<\dagger\dagger, \dagger\dagger, \mathbf{W}>\}$$

this composition has the value

$$\left(\boxed{\texttt{VAR Ch: CHAR}} \circ \boxed{\texttt{BEGIN END}} \circ \boxed{\texttt{VAR Ch: CHAR}}^{\mathrm{T}}\right)$$
$$\left(\{\texttt{INPUT}\cdot<\dagger\dagger, \nabla\,/, \mathbf{R}>,\ \texttt{OUTPUT}\cdot<\dagger\dagger, \dagger\dagger, \mathbf{W}>\}\right)$$
$$= \left(\boxed{\texttt{BEGIN END}} \circ \boxed{\texttt{VAR Ch: CHAR}}^{\mathrm{T}}\right)$$
$$\left(\{\texttt{INPUT}\cdot<\dagger\dagger, \nabla\,/, \mathbf{R}>,\ \texttt{OUTPUT}\cdot<\dagger\dagger, \dagger\dagger, \mathbf{W}>,\ \texttt{Ch}\cdot?\}\right)$$
$$= \boxed{\texttt{VAR Ch: CHAR}}^{\mathrm{T}}\left(\{\texttt{INPUT}\cdot<\dagger\dagger, \nabla\,/, \mathbf{R}>,\ \texttt{OUTPUT}\cdot<\dagger\dagger, \dagger\dagger, \mathbf{W}>,\ \texttt{Ch}\cdot?\}\right)$$
$$= \{\texttt{INPUT}\cdot<\dagger\dagger, \nabla\,/, \mathbf{R}>,\ \texttt{OUTPUT}\cdot<\dagger\dagger, \dagger\dagger, \mathbf{W}>\}$$

That is, as expected, $\boxed{\texttt{VAR Ch: CHAR}}$ contains all pairs of the form

$<s, s \cup \{\texttt{Ch}\cdot c\}>$ for character c

(which we abbreviate to adding Ch·? to the state); $\boxed{\texttt{BEGIN END}}$ does not alter the state; and

$\boxed{\texttt{VAR Ch: CHAR}}^{\mathrm{T}}$ removes the pair abbreviated as Ch·?.

The general case of a block's part meaning follows the example above closely. When there are no declarations, the block meaning is just the meaning of its BEGIN statement. When there is a VAR or PROCEDURE declaration F followed by a block B, the meaning is:

$$\boxed{F\ ;\ B} = \boxed{F} \circ \boxed{B} \circ \boxed{F}^{\mathrm{T}}.$$

A block with a single variable declaration and a single procedure declaration would be analyzed as follows:

$$\boxed{\texttt{VAR...; PROCEDURE...; BEGIN...END}}$$
$$= \boxed{\texttt{VAR...}} \circ \boxed{\texttt{PROCEDURE..; BEGIN...END}} \circ \boxed{\texttt{VAR...}}^{\mathrm{T}}$$
$$= \boxed{\texttt{VAR...}} \circ \boxed{\texttt{PROCEDURE...}} \circ \boxed{\texttt{BEGIN...END}} \circ \boxed{\texttt{PROCEDURE...}}^{\mathrm{T}} \circ \boxed{\texttt{VAR...}}^{\mathrm{T}}.$$

The part meaning of a block is typical of the program calculus. A substring of a program corresponding to a syntactic part is further divided into components. Then the meaning of the original substring is defined as in terms of its components' meanings.

The remainder of this chapter defines the part meanings for statements of CF Pascal. But by considering statement actions in execution tables it is possible to anticipate the formal definitions, and work out a complete example of the program calculus. Consider the program Q:

```
PROGRAM CopyChar(INPUT,OUTPUT);
  VAR
    Ch: CHAR;
  BEGIN
    READ(Ch);
    WRITELN(Ch)
  END.
```

$\boxed{Q}$ can be calculated for input the 1-list <†**ABC**†> as:

$$\boxed{Q}(\texttt{<†ABC†>}) = \boxed{\texttt{PROGRAM ...END.}}(\texttt{<†ABC†>})$$
$$= (\boxed{\texttt{PROGRAM...}} \circ \boxed{\texttt{VAR...END}} \circ \boxed{\texttt{.}})(\texttt{<†ABC†>}) \quad (1)$$
$$= (\boxed{\texttt{VAR...END}} \circ \boxed{\texttt{.}})\,(\{\texttt{INPUT·<††,ABC/,R>, OUTPUT·<††,††,W>}\}) \quad (2)$$
$$= (\boxed{\texttt{VAR...}} \circ \boxed{\texttt{BEGIN...END}} \circ \boxed{\texttt{VAR...}}^{\mathrm{T}} \circ \boxed{\texttt{.}})$$
$$\quad (\{\texttt{INPUT·<††,ABC/,R>, OUTPUT·<††,††,W>}\}) \quad (3)$$
$$= (\boxed{\texttt{BEGIN...END}} \circ \boxed{\texttt{VAR...}}^{\mathrm{T}} \circ \boxed{\texttt{.}})$$
$$\quad (\{\texttt{INPUT·<††,ABC/,R>, OUTPUT·<††,††,W>, Ch·?}\}) \quad (4)$$
$$= (\boxed{\texttt{VAR...}}^{\mathrm{T}} \circ \boxed{\texttt{.}})\,(\{\texttt{INPUT·<†A†,BC/,R>, OUTPUT·<A/,††,W>, Ch·A}\}) \quad (5)$$
$$= \boxed{\texttt{.}}(\{\texttt{INPUT·<†A†,BC/,R>, OUTPUT·<A/,††,W>}\}) \quad (6)$$
$$= \texttt{<†A†>} \quad (7)$$

Step 1 expands the program meaning $\boxed{Q}$ by definition into the composition of its header, block, and period meanings. Step 2 records the value of the first header meaning of the composition. Step 3 expands the block meaning into the composition of its declarations, BEGIN statement, and declaration transpose meanings. Step 4 records the value of the first declaration meaning of the composition. Step 5 records the value of the BEGIN statement meaning (by direct inspection). Step 6 records the value of the declaration transpose meaning. Step 7 records the value of the final period meaning.

The domains and ranges of the part meanings introduced so far are given in the following table:

Table of Domains and Ranges of Part Meanings

Program part	*Domain*	*Range*
program	lists of strings	lists of strings
program header	lists of strings	execution states
final period	execution states	lists of strings
block	execution states	execution states
declaration	execution states	execution states
BEGIN statement	execution states	execution states

6.1.6 Exercises

6.1.1 Write out in full detail the final execution state (just before the end of execution) in the partial execution table for Program 22 given in Section 1.5.1.

6.1.2 If P is the program:

```
PROGRAM Read2 (INPUT,OUTPUT);
  VAR
    C1, C2: CHAR;
  BEGIN
    READ(C2);
    READ(C1);
    WRITELN(C1, C2, 'REV')
  END.
```

a) What is $\boxed{P}$ by direct inspection?

b) What is the execution state (in the abbreviated notation) just after the second READ statement has been executed, for input <†**ABC**†>?

6.1.3 If $r \circ r^{\mathrm{T}} = I$, where I is the identity over domain(r), is it always the case that $r^{\mathrm{T}} \circ r = I$? Explain.

6.1.4

a) Give an example of a relation that is not a function, but its transpose is a function.

b) Give an example of a function whose transpose is a relation that is not a function.

6.1.5 Is the transpose of a function or relation always its right inverse? Why or why not? (Hint: the results of this section show that only if the transpose is *not* a function, could it fail to be the inverse.)

6.1.6 In the definition of $\boxed{\texttt{VAR ...}}$, identifier-value pairs are added with set union to the argument state to form the result state. Explain why it *never* happens in actual programs that the added identifiers are already in the argument state.

6.1.7 Suppose that an execution state s contains the identifier-value pair Ch·**F** (the result of an assignment, as the next section shows). Carefully explain how

$$\boxed{\texttt{VAR Ch: CHAR}}^{\mathrm{T}}(s)$$

is computed. Do not make use of the ?-abbreviation.

6.1.8 Given program R:

```
PROGRAM PrintA (INPUT,OUTPUT);
  VAR
    Ch1, Ch2: CHAR;
  BEGIN
    READ(Ch1);
    WRITELN('A')
  END.
```

Work out the program meaning of R for input <†**XYZ**†>, identifying intermediate execution states determined by the VAR part meaning and its transpose. Obtain the part meaning of the BEGIN statement by direct inspection.

6.1.9 Describe the relation defined by

$$[.] \circ [\texttt{PROGRAM Any (INPUT, OUTPUT)}]$$

(i.e., give its domain and range, and what values it takes; is it a function?).

6.1.10 Given program S:

```
PROGRAM PrintB(INPUT,OUTPUT);
  PROCEDURE WriteB;
    BEGIN
      WRITELN('B')
    END;
  BEGIN {PrintB}
    WriteB
  END.
```

Work out the program meaning of S for input <†**AB**†>, identifying intermediate execution states determined by the procedure part meaning and its transpose. Determine the part meaning of the BEGIN statement by direct inspection.

6.2 The Meaning of Sequential Statements

Preview

This section defines the meanings of the CF Pascal sequential statements: assignment statements, null statements, BEGIN statements, WRITE statements, READ statements, and simple procedure statements.

New ideas: semantics for statements

Sequential programs have no IF or WHILE statements, so control passes directly from one statement to the next in sequence. Each kind of CF Pascal sequential statement will have a part meaning defined for it, a transformation from execution states to execution states. The box notation symbolizes the part meaning of each statement.

6.2.1 Assignment Statements

The meaning of an assignment statement is a function from execution states to execution states. In an execution table, an assignment statement causes the left-side identifier to be associated with a new value, obtained from the expression on the right side.

The box notation can be extended to give meaning to the expressions that appear on the right side of assignments. The part meaning of such an expression is the value taken for a given execution state. If the assignment statement is

```
Ch1 := Ch2
```

the expression on the right is a variable, and its value is directly available in the current execution state. For example, in a state containing the pair Ch2·**B** the value is **B**. Thus for an identifier W

declared CHAR, the part meaning of the expression W in state s is the value $W(s)$ paired with W, that is,

$$\boxed{W} = \{\langle s, s(W)\rangle\}.$$

For example,

$$\boxed{\texttt{Ch}}(\{\texttt{INPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Ch}\cdot\mathbf{F},\ \texttt{V1}\cdot\mathbf{C}\})$$
$$= \{\texttt{INPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Ch}\cdot\mathbf{F},\ \texttt{V1}\cdot\mathbf{C}\}(\texttt{Ch})$$
$$= \mathbf{F}.$$

Values of literal character expressions are easier to calculate, for they do not depend on the execution state at all. No matter what may be in that state, the literal itself gives the value. For example:

$$\boxed{\texttt{'A'}}(s) = \mathbf{A}$$

$$\boxed{\texttt{'B'}}(s) = \mathbf{B}$$

and so on.

This definition of a part meaning for character expressions leads to a concise definition for the part meaning of an assignment statement with left-side identifier V and right-side expression E:

$$\boxed{V\ :=\ E} = \{\langle s, t\rangle\colon t = (s - \{\langle V, c\rangle\colon c \text{ is a character}\}) \cup \{\langle V, \boxed{E}(s)\rangle\}\}.$$

The set difference removes any old value for V, then the union adds the appropriate new value. Using sample identifiers V1 and V2 and the literal character expression 'A', the definition becomes:

$$\boxed{\texttt{V1 := V2}} = \{\langle s, t\rangle\colon t \text{ is the same as } s \text{ except that } \boxed{\texttt{V1}}(t) = \boxed{\texttt{V2}}(s)\}$$
$$\boxed{\texttt{V1 := 'A'}} = \{\langle s, t\rangle\colon t \text{ is the same as } s \text{ except that } \boxed{\texttt{V1}}(t) = \mathbf{A}\}.$$

Here are five examples:

$$\boxed{\texttt{V1 := 'B'}}(\{\texttt{V1}\cdot\mathbf{A}, \ldots\}) = \{\texttt{V1}\cdot\mathbf{B}, \ldots\},$$
$$\boxed{\texttt{V2 := V1}}(\{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot\mathbf{B}, \ldots\}) = (\{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot\mathbf{A}, \ldots\}),$$
$$\boxed{\texttt{V1 := V1}}(\{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot\mathbf{B}, \ldots\}) = (\{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot\mathbf{B}, \ldots\}),$$
$$\boxed{\texttt{V1 := V2}}(\{\texttt{V1}\cdot?,\ \texttt{V2}\cdot\mathbf{A}, \ldots\}) = (\{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot\mathbf{A}, \ldots\}),$$
$$\boxed{\texttt{V1 := V2}}(\{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot?, \ldots\}) = (\{\texttt{V1}\cdot?,\ \texttt{V2}\cdot?, \ldots\}).$$

The domain and range elements for the part meanings introduced so far are given in the following table:

Table of Domains and Ranges of Part Meanings

Program part	*Domain*	*Range*
program	lists of strings	lists of strings
program header	lists of strings	execution states
final period	execution states	lists of strings
block	execution states	execution states
declaration	execution states	execution states
statement	execution states	execution states
expression	execution states	characters

6.2.2 Null Statements

A null statement "does nothing," which means that whatever execution state exists before its execution is unchanged afterwards. The meaning with this behavior is an identity function I. Thus,

$$[\,] = I = \{<s, s>\}.$$

6.2.3 BEGIN Statements

A BEGIN statement must contain one or more constituent statements, separated by semicolons. There must be at least one statement (even

```
BEGIN
END
```

contains a null statement). The first statement is either the only statement of a BEGIN statement, and there will be no semicolons; or it will be the statement preceding the first semicolon. If there is a semicolon, there is a statement following it, if only a null statement.

The meaning of the BEGIN statement can be defined a constituent statement at a time. Let S be a single statement and T be a sequence of one or more statements separated by semicolons:

$$[\![\texttt{BEGIN } S \texttt{ END}]\!] = [\![S]\!]$$

$$[\![\texttt{BEGIN } S \texttt{ ; } T \texttt{ END}]\!] = [\![S]\!] \circ [\![\texttt{BEGIN } T \texttt{ END}]\!].$$

For example:

$$[\![\texttt{BEGIN V1 := V2; V2 := V3 END}]\!]$$
$$= [\![\texttt{V1 := V2}]\!] \circ [\![\texttt{BEGIN V2 := V3 END}]\!] = [\![\texttt{V1 := V2}]\!] \circ [\![\texttt{V2 := V3}]\!],$$

and

$$[\![\texttt{BEGIN V1 := V2; V2 := V3; V3 := V4 END}]\!]$$
$$= [\![\texttt{V1 := V2}]\!] \circ [\![\texttt{BEGIN V2 := V3; V3 := V4 END}]\!]$$
$$= [\![\texttt{V1 := V2}]\!] \circ [\![\texttt{V2 := V3}]\!] \circ [\![\texttt{V3 := V4}]\!].$$

BEGIN statements can change values of several variables. For example:

$$[\![\texttt{BEGIN V1 := 'A'; V2 := 'B' END}]\!](\{\texttt{V1}\cdot\mathbf{X},\ \texttt{V2}\cdot\mathbf{V}, \ldots\})$$
$$= ([\![\texttt{V1 := 'A'}]\!] \circ [\![\texttt{V2 := 'B'}]\!])(\{\texttt{V1}\cdot\mathbf{X},\ \texttt{V2}\cdot\mathbf{V}, \ldots\}) \quad (1)$$
$$= [\![\texttt{V2 := 'B'}]\!]([\![\texttt{V1 := 'A'}]\!](\{\texttt{V1}\cdot\mathbf{X},\ \texttt{V2}\cdot\mathbf{V}, \ldots\}) \quad (2)$$
$$= [\![\texttt{V2 := 'B'}]\!](\{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot\mathbf{V}, \ldots\}) \quad (3)$$
$$= (\{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot\mathbf{B}, \ldots\}). \quad (4)$$

Step 1 expands the part meaning of the BEGIN statement into the composition of its constituent statements. Step 2 converts to the value notation, step 3 applies the first constituent meaning, and step 4 the second constituent meaning.

BEGIN statements can also change values of one variable several times. The only visible effect is the last change. For example:

$$[\![\texttt{BEGIN V1 := 'A'; V1 := 'B' END}]\!](\{\texttt{V1}\cdot\mathbf{H},\ \texttt{V2}\cdot\mathbf{K}, \ldots\})$$
$$= ([\![\texttt{V1 := 'A'}]\!] \circ [\![\texttt{V1 := 'B'}]\!])(\{\texttt{V1}\cdot\mathbf{H},\ \texttt{V2}\cdot\mathbf{K}, \ldots\})$$
$$= [\![\texttt{V1 := 'B'}]\!](\{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot\mathbf{K}, \ldots\})$$
$$= \{\texttt{V1}\cdot\mathbf{B},\ \texttt{V2}\cdot\mathbf{K}, \ldots\},$$

and in this case, no trace is left that V1 ever had value **A**.

Moving a two-character window from V1,V2 to V2,V3 can be analyzed as follows:

$$\boxed{\texttt{BEGIN V3 := V2; V2 := V1 END}}(\{\texttt{V1}\cdot\mathbf{X},\ \texttt{V2}\cdot\mathbf{Y},\ \texttt{V3}\cdot\mathbf{Z},\ \ldots\})$$
$$= (\boxed{\texttt{V3 := V2}} \circ \boxed{\texttt{V2 := V1}})\,(\{\texttt{V1}\cdot\mathbf{X},\ \texttt{V2}\cdot\mathbf{Y},\ \texttt{V3}\cdot\mathbf{Z},\ \ldots\})$$
$$= \boxed{\texttt{V2 := V1}}(\{\texttt{V1}\cdot\mathbf{X},\ \texttt{V2}\cdot\mathbf{Y},\ \texttt{V3}\cdot\mathbf{Y},\ \ldots\})$$
$$= \{\texttt{V1}\cdot\mathbf{X},\ \texttt{V2}\cdot\mathbf{X},\ \texttt{V3}\cdot\mathbf{Y},\ \ldots\}.$$

The familiar pattern used to exchange the values of two variables V1 and V2 can be analyzed as a composition of three meanings as follows:

$$\boxed{\texttt{BEGIN V1 := V2; V2 := V3; V3 := V1 END}}(\{\texttt{V1}\cdot\mathbf{X},\ \texttt{V2}\cdot\mathbf{Y},\ \texttt{V3}\cdot\mathbf{Z},\ \ldots\})$$
$$= (\boxed{\texttt{V1 := V2}} \circ \boxed{\texttt{V2 := V3}} \circ \boxed{\texttt{V3 := V1}})\,(\{\texttt{V1}\cdot\mathbf{X},\ \texttt{V2}\cdot\mathbf{Y},\ \texttt{V3}\cdot\mathbf{Z},\ \ldots\})$$
$$= (\boxed{\texttt{V2 := V3}} \circ \boxed{\texttt{V3 := V1}})\,(\{\texttt{V1}\cdot\mathbf{Y},\ \texttt{V2}\cdot\mathbf{Y},\ \texttt{V3}\cdot\mathbf{Z},\ \ldots\})$$
$$= \boxed{\texttt{V3 := V1}}(\{\texttt{V1}\cdot\mathbf{Y},\ \texttt{V2}\cdot\mathbf{Z},\ \texttt{V3}\cdot\mathbf{Z},\ \ldots\})$$
$$= \{\texttt{V1}\cdot\mathbf{Y},\ \texttt{V2}\cdot\mathbf{Z},\ \texttt{V3}\cdot\mathbf{Y},\ \ldots\}.$$

6.2.4 WRITE Statements

Files have values that are 3-lists, and the actions of READ and WRITE statements have been precisely described in Section 5.2.2. It is not difficult to translate that description into formal meaning—the statements' effect on execution states. Let f be a file identifier, and e a character expression.

$$\boxed{\texttt{REWRITE(}\ f\ \texttt{)}} = \{\langle s, t\rangle\colon t = (s - \{\langle f, u\rangle\colon u \text{ is a 3-list}\}) \cup \{\langle f, \langle \dagger\dagger, \dagger\dagger, \mathbf{W}\rangle\rangle\}\}.$$

$$\boxed{\texttt{WRITE(}\ f\ \texttt{,}\ e\ \texttt{)}} = \{\langle s, t\rangle\colon s(f) = \langle x, \dagger\dagger, \mathbf{W}\rangle \text{ where } x \text{ is a string},$$
$$t = (s - \{\langle f, u\rangle\colon u \text{ is a 3-list}\}) \cup \{\langle f, \langle x \nabla \boxed{e}(s), \dagger\dagger, \mathbf{W}\rangle\rangle\}\}.$$

$$\boxed{\texttt{WRITELN(}\ f\ \texttt{)}} = \{\langle s, t\rangle\colon s(f) = \langle x, \dagger\dagger, \mathbf{W}\rangle \text{ where } x \text{ is a string},$$
$$t = (s - \{\langle f, u\rangle\colon u \text{ is a 3-list}\}) \cup \{\langle f, \langle x \nabla /, \dagger\dagger, \mathbf{W}\rangle\rangle\}\}.$$

When the file identifier is missing, OUTPUT is the file for WRITE and WRITELN statements, for example,

$$\boxed{\texttt{WRITELN}} = \{\langle s, t\rangle\colon s(\texttt{OUTPUT}) = \langle x, \dagger\dagger, z\rangle \text{ where } x \text{ is a string},$$
$$t = (s - \{\langle \texttt{OUTPUT}, u\rangle\colon u \text{ is a 3-list}\}) \cup \{\langle \texttt{OUTPUT}, \langle x \nabla /, \dagger\dagger, \mathbf{W}\rangle\rangle\}\}.$$

For example:

$$\boxed{\texttt{REWRITE (F1)}}(\{\texttt{F1}\cdot\langle\mathbf{AXB},\dagger\dagger,\mathbf{W}\rangle, \ldots\}) = \{\texttt{F1}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle, \ldots\},$$
$$\boxed{\texttt{WRITE (F1, Ch)}}(\{\texttt{F1}\cdot\langle\mathbf{AB},\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Ch}\cdot\mathbf{C}, \ldots\}) = \{\texttt{F1}\cdot\langle\mathbf{ABC},\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Ch}\cdot\mathbf{C}, \ldots\},$$
$$\boxed{\texttt{WRITELN}}(\{\texttt{OUTPUT}\cdot\langle\mathbf{Line1},\dagger\dagger,\mathbf{W}\rangle, \ldots\}) = \{\texttt{OUTPUT}\cdot\langle\mathbf{Line1/},\dagger\dagger,\mathbf{W}\rangle, \ldots\}.$$

WRITE statements involving several expressions were defined as combinations of simpler statements; composition of meanings is just the right notion to use in these definitions. For example,

$$\boxed{\texttt{WRITE (Ch, 'A')}} = \boxed{\texttt{WRITE (Ch)}} \circ \boxed{\texttt{WRITE ('A')}},$$
$$\boxed{\texttt{WRITE ('AB')}} = \boxed{\texttt{WRITE ('A')}} \circ \boxed{\texttt{WRITE ('B')}},$$
$$\boxed{\texttt{WRITELN (F1, Ch)}} = \boxed{\texttt{WRITE (F1, Ch)}} \circ \boxed{\texttt{WRITELN (F1)}}.$$

Enough of the program calculus has been presented to handle in complete detail programs like the first ones seen in Chapter 1. For example, consider

```
PROGRAM WriteHello (INPUT,OUTPUT);
  VAR
    LetterL: CHAR;
  BEGIN
    LetterL := 'L';
    WRITELN ('H', 'E', LetterL, LetterL, 'O')
  END.
```

Successive calculations reveal the meaning of this program, applied to the input line <**ABC**>:

$$\boxed{\texttt{PROGRAM WriteHello ... END.}}(\langle\mathbf{ABC}\rangle)$$

$$= (\boxed{\texttt{PROGRAM ...}} \circ \boxed{\texttt{VAR ... BEGIN ... END}} \circ \boxed{.})(\langle\mathbf{ABC}\rangle)$$

$$= (\boxed{\texttt{VAR ... BEGIN ... END}} \circ \boxed{.})\,(\{\texttt{INPUT}\cdot\langle\dagger\dagger,\mathbf{ABC}/,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle\})$$

$$= (\boxed{\texttt{VAR...}} \circ \boxed{\texttt{BEGIN ... END}} \circ \boxed{\texttt{VAR...}}^{\mathrm{T}} \circ \boxed{.})$$
$$(\{\texttt{INPUT}\cdot\langle\dagger\dagger,\mathbf{ABC}/,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle\})$$

$$= (\boxed{\texttt{BEGIN ... END}} \circ \boxed{\texttt{VAR...}}^{\mathrm{T}} \circ \boxed{.})$$
$$(\{\texttt{INPUT}\cdot\langle\dagger\dagger,\mathbf{ABC}/,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{LetterL}\cdot?\})$$

$$= (\boxed{\texttt{LetterL := 'L'}}$$
$$\circ \boxed{\texttt{WRITELN('H', 'E', LetterL, LetterL, 'O')}} \circ \boxed{\texttt{VAR...}}^{\mathrm{T}} \circ \boxed{.})$$
$$(\{\texttt{INPUT}\cdot\langle\dagger\dagger,\mathbf{ABC}/,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{LetterL}\cdot?\})$$

$$= (\boxed{\texttt{WRITELN('H', 'E', LetterL, LetterL, 'O')}} \circ \boxed{\texttt{VAR...}}^{\mathrm{T}} \circ \boxed{.})$$
$$(\{\texttt{INPUT}\cdot\langle\dagger\dagger,\mathbf{ABC}/,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{LetterL}\cdot\mathbf{L}\})$$

$$= (\boxed{\texttt{VAR...}}^{\mathrm{T}} \circ \boxed{.})\,(\{\texttt{INPUT}\cdot\langle\dagger\dagger,\mathbf{ABC}/,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\mathbf{HELLO}/,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{LetterL}\cdot\mathbf{L}\})$$

$$= \boxed{.}\,(\{\texttt{INPUT}\cdot\langle\dagger\dagger,\mathbf{ABC}/,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\mathbf{HELLO}/,\dagger\dagger,\mathbf{W}\rangle\})$$

$$= \langle\mathbf{HELLO}\rangle.$$

Thus the meaning of the program on this input is <**HELLO**>:

$$\boxed{\texttt{PROGRAM WriteHello ... END.}}(\langle\mathbf{ABC}\rangle) = \langle\mathbf{HELLO}\rangle.$$

6.2.5 READ Statements

The actions of READ statements given in Section 5.2.2 can also be translated into mappings of execution states. Let f be a file identifier and c a CHAR variable. Then:

$$\boxed{\texttt{RESET(}\ f\ \texttt{)}} = \{\langle s, t\rangle\colon s(f) = \langle x, y, \mathbf{R}\rangle \text{ where } x, y \text{ are strings},$$
$$t = (s - \{\langle f, u\rangle\colon u \text{ is a 3-list}\}) \cup \{\langle f, \langle\dagger\dagger, x \,\&\, y, \mathbf{R}\rangle\rangle\}\}.$$

$$\boxed{\texttt{READ(}\ f\ \texttt{,}\ c\ \texttt{)}} = \{\langle s, t\rangle\colon s(f) = \langle x, y, \mathbf{R}\rangle, \text{ where } x, y \text{ are strings}, y \neq \dagger\dagger,$$
$$t = ((s - \{\langle f, u\rangle\colon u \text{ is a 3-list}\}) - \{\langle c, v\rangle\colon v \text{ is a character}\})$$
$$\cup \{\langle f, \langle x\ \nabla(\Theta\, y), \Lambda\, y, \mathbf{R}\rangle\rangle\}$$
$$\cup \{\langle c, \Theta\, y\rangle\colon \Theta\, y \neq /\} \cup \{\langle c, \square\rangle\colon \Theta\, y = /\}\}.$$

$$\boxed{\texttt{READLN(}\ f\ \texttt{)}} = \{\langle s, t\rangle\colon s(f) = \langle x, y, \mathbf{R}\rangle, \text{ where } x, y \text{ are strings},$$
$$y = (j\ \nabla\ /)\ \&\ k,\ / \text{ not a substring of } j,$$
$$t = (s - \{\langle f, u\rangle\colon u \text{ is a 3-list}\}) \cup \{\langle f, \langle x \,\&\, (j\ \nabla\ /), k, \mathbf{R}\rangle\rangle\}\}.$$

When the file identifier is missing, INPUT is the file for READ and READLN statements, for example,

$$\boxed{\texttt{READ(c)}} = \{\langle s, t\rangle : s(\texttt{INPUT}) = \langle x, y, \mathbf{R}\rangle, \text{ where } x, y \text{ are strings, } y \neq \dagger\dagger,$$
$$t = ((s - \{\langle \texttt{INPUT}, u\rangle : u \text{ is a 3-list}\}) - \{\langle c, v\rangle : v \text{ is a character}\})$$
$$\cup \{\langle \texttt{INPUT}, \langle x\ \nabla(\Theta\ y), \Lambda\ y, \mathbf{R}\rangle\rangle\}$$
$$\cup \{\langle c, \Theta\ y\rangle : \Theta\ y \neq /\} \cup \{\langle c, \square\rangle : \Theta\ y = /\}\}.$$

For example:

$$\boxed{\texttt{RESET(F1)}}(\{\texttt{F1}\cdot\langle\mathbf{AB},\mathbf{CD}/,\mathbf{R}\rangle, \ldots\}) = \{\texttt{F1}\cdot\langle\dagger\dagger,\mathbf{ABCD}/,\mathbf{R}\rangle, \ldots\},$$

$$\boxed{\texttt{READ(F1, Ch)}}(\{\texttt{F1}\cdot\langle\mathbf{AB},\mathbf{CD}/,\mathbf{R}\rangle, \texttt{Ch}\cdot\mathbf{Q}, \ldots\})$$
$$= (\{\texttt{F1}\cdot\langle\mathbf{ABC},\mathbf{D}/,\mathbf{R}\rangle, \texttt{Ch}\cdot\mathbf{C}, \ldots\}),$$

$$\boxed{\texttt{READLN}}(\{\texttt{INPUT}\cdot\langle\mathbf{AB},\mathbf{CD}/,\mathbf{R}\rangle, \ldots\}) = (\{\texttt{INPUT}\cdot\langle\mathbf{ABCD}/,\dagger\dagger,\mathbf{R}\rangle, \ldots\}).$$

More complicated READ statement meanings are defined as compositions, for example:

$$\boxed{\texttt{READ(Ch1, Ch2)}} = \boxed{\texttt{READ(Ch1)}} \circ \boxed{\texttt{READ(Ch2)}},$$

$$\boxed{\texttt{READLN(Ch)}} = \boxed{\texttt{READ(Ch)}} \circ \boxed{\texttt{READLN}}.$$

As an example, consider the program:

```
PROGRAM Change2 (INPUT,OUTPUT);
  VAR
    C2: CHAR;
  BEGIN
    READ (C2);
    WRITE (C2);
    READ (C2);
    WRITELN ('2')
  END.
```

In calculating $\boxed{\texttt{PROGRAM Change2 ... END.}}(\langle\dagger\mathbf{ABC}\dagger\rangle)$, the first steps are the same as those for `WriteHello` above:

$$\boxed{\texttt{PROGRAM Change2 ... END.}}(\langle\dagger\mathbf{ABC}\dagger\rangle) =$$
$$(\boxed{\texttt{PROGRAM Change2 ...}} \circ \boxed{\texttt{VAR...}} \circ \boxed{\texttt{BEGIN ... END}} \circ \boxed{\texttt{VAR...}}^{\mathrm{T}} \circ \boxed{.})\ (\langle\dagger\mathbf{ABC}\dagger\rangle)$$
$$\ldots$$
$$= (\boxed{\texttt{BEGIN ... END}} \circ \boxed{\texttt{VAR...}}^{\mathrm{T}} \circ \boxed{.})$$
$$(\{\texttt{INPUT}\cdot\langle\dagger\dagger,\mathbf{ABC}/,\mathbf{R}\rangle, \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle, \texttt{C2}\cdot?\})$$
$$= (\boxed{\texttt{READ(C2)}} \circ \boxed{\texttt{WRITE(C2)}} \circ \boxed{\texttt{READ(C2)}} \circ \boxed{\texttt{WRITELN('2')}} \circ \boxed{\texttt{VAR ...}}^{\mathrm{T}} \circ \boxed{.})$$
$$(\{\texttt{INPUT}\cdot\langle\dagger\dagger,\mathbf{ABC}/,\mathbf{R}\rangle, \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle, \texttt{C2}\cdot?\})$$
$$= (\boxed{\texttt{WRITE(C2)}} \circ \boxed{\texttt{READ(C2)}} \circ \boxed{\texttt{WRITELN('2')}} \circ \boxed{\texttt{VAR ...}}^{\mathrm{T}} \circ \boxed{.})$$
$$(\{\texttt{INPUT}\cdot\langle\dagger\mathbf{A}\dagger,\mathbf{BC}/,\mathbf{R}\rangle, \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle, \texttt{C2}\cdot\mathbf{A}\})$$
$$= (\boxed{\texttt{READ(C2)}} \circ \boxed{\texttt{WRITELN('2')}} \circ \boxed{\texttt{VAR ...}}^{\mathrm{T}} \circ \boxed{.})$$
$$(\{\texttt{INPUT}\cdot\langle\dagger\mathbf{A}\dagger,\mathbf{BC}/,\mathbf{R}\rangle, \texttt{OUTPUT}\cdot\langle\dagger\mathbf{A}\dagger,\dagger\dagger,\mathbf{W}\rangle, \texttt{C2}\cdot\mathbf{A}\})$$
$$= (\boxed{\texttt{WRITELN('2')}} \circ \boxed{\texttt{VAR ...}}^{\mathrm{T}} \circ \boxed{.})$$
$$(\{\texttt{INPUT}\cdot\langle\mathbf{AB},\mathbf{C}/,\mathbf{R}\rangle, \texttt{OUTPUT}\cdot\langle\dagger\mathbf{A}\dagger,\dagger\dagger,\mathbf{W}\rangle, \texttt{C2}\cdot\mathbf{B}\})$$
$$= (\boxed{\texttt{VAR ...}}^{\mathrm{T}} \circ \boxed{\dagger.\dagger})\ (\{\texttt{INPUT}\cdot\langle\mathbf{AB},\mathbf{C}/,\mathbf{R}\rangle, \texttt{OUTPUT}\cdot\langle\mathbf{A2},\dagger\dagger,\mathbf{W}\rangle, \texttt{C2}\cdot\mathbf{B}\})$$
$$= \boxed{.}(\{\texttt{INPUT}\cdot\langle\mathbf{AB},\mathbf{C}/,\mathbf{R}\rangle, \texttt{OUTPUT}\cdot\langle\mathbf{A2},\dagger\dagger,\mathbf{W}\rangle\})$$
$$= \langle\dagger\mathbf{A2}\dagger\rangle.$$

Thus

$\boxed{\texttt{PROGRAM Change2 ... END.}}(\langle\dagger\textbf{ABC}\dagger\rangle) = \langle\dagger\textbf{A2}\dagger\rangle$.

6.2.6 Simple Procedure Statements

The declaration of a CHAR variable adds its identifier paired with an unknown value to the state, so that later the value can be changed (with assignment or READ statements), or obtained for use in evaluating expressions. A procedure declaration also adds the identifier to the state, but paired with the full text of the procedure <body>. Later, this <body> text supplies the meaning for a procedure statement. For example, in the program,

```
PROGRAM SimpleProc(INPUT,OUTPUT);
  PROCEDURE Demo;
    BEGIN
      WRITELN('Invoked.')
    END;
  BEGIN
    Demo
  END.
```

the procedure declaration adds the pair

$\langle\dagger$**Demo**$\dagger$, $\dagger$**BEGIN WRITELN('Invoked.') END**$\dagger\rangle$

to the state, and the subsequent procedure statement has the meaning

$\boxed{\texttt{Demo}} = \boxed{\texttt{BEGIN WRITELN('Invoked.') END}}$.

The formal definition of procedure statement part meaning follows the example exactly. Let N be a procedure identifier. Then:

$\boxed{N} = \{\langle s, \boxed{s\,(N)}(s)\rangle\}$.

6.2.7 Exercises

6.2.1 For the program in each part, work out the program meaning for input $\langle\dagger$**ABCD**$\dagger\rangle$, showing all intermediate states.

a) VERTICALGREETING (Program 7, Section 1.2).

b) CopyTwice (Program 13, Section 1.3).

c) CopyReversedToo (Program 16, Section 1.3.2).

6.2.2 For the program in each part, work out the program meaning for input $\langle\dagger$**A**$\dagger\rangle$, showing all intermediate states.

a) CopyReversedToo (Program 16, Section 1.3.2).

b) CopyAndShuffle (Program 17, Section 1.3.2).

6.2.3 Work out the program meaning for SimpleProc given in Section 6.2.6, giving each intermediate state, when the input is an arbitrary string x. Can you write the result without mentioning x?

6.2.4 Let P be the program

```
PROGRAM AssignThem(INPUT, OUTPUT);
  VAR
    Ch1, Ch2: CHAR;
  BEGIN
    Ch2 := Ch1
  END.
```

Explain why the formal definitions make this program meaning a constant function:

$[\![P]\!] = \langle\dagger\dagger\rangle$.

It seems intuitively reasonable to some people that the definition for this program *should* be:

$[\![P]\!] = \{\}$,

that is, the empty function.

a) Give an argument why the definition as a constant function is right.

b) Give an argument why the definition as the empty function is right.

The lesson is that although definitions are always "true" (by definition!), there can be honest dispute about how to make them correspond with intuition. But don't forget which *is* the definition that you must use.

6.2.5 If a sequential program S contains two READ statements, then $[\![S]\!]$ is *never* defined for all input strings. Explain which strings are not in domain($[\![S]\!]$) and why not.

6.3 The Analysis of Sequential Statements

Preview

The meaning of a BEGIN statement with constituent sequential statements can be summarized by a concurrent assignment that lists the variables and their final values. Symbolic execution recorded in trace tables can be used to derive the meanings of BEGIN statements as concurrent assignments or to verify the correctness of meanings given independently.

New ideas: concurrent assignments, symbolic execution, trace tables.

The program calculus has the goal of calculating the meaning $[\![P]\!]$ of any program P. The principle is simple, and the definitions of meaning for program parts, along with composition, allow working out the meaning for an entire program. In practice, however, using the definitions directly may be a long and tedious procedure. In this section, some shorthand methods are given that make practical program analysis easier.

6.3.1 Concurrent Assignments

BEGIN statements can assign values to more than one variable. The ability to summarize the effect of a BEGIN statement is so important that it will be made into a formal comment in CF Pascal, called a *concurrent assignment*. The Pascal machine does not have the ability to execute concurrent assignments—they are not part of Pascal syntax. However, the form of the comment is a simple extension of the CF Pascal assignment statement. A list of variables is written on the left side of the assignment operator, and a list of expressions on the right side, these two lists being of equal length. A concurrent assignment expresses the knowledge that the expressions, computed all at the same time, are the values taken by the corresponding variables. For example, the effect of the BEGIN statement:

```
BEGIN
  V1 := 'A';
  V2 := 'B'
END
```

is given by the concurrent assignment

```
<V1,V2> := <'A','B'>
```

while the BEGIN statement

```
BEGIN
  V1 := 'A';
  V1 := 'B'
END
```

has the effect given by:

```
<V1> := <'B'>.
```

When no confusion can arise, the angle brackets of the lists are omitted; for example:

```
V1,V2 := 'A','B'
```

in the first example above. Concurrent assignments make excellent comments, because they express exactly what the result of a BEGIN statement must be. For example,

```
BEGIN {V1 := 'B'}
  V1 := 'A';
  V1 := 'B'
END
```

assures the reader that this peculiar construction is intended, not a typing mistake.

For any characters x, y, z,

$$\boxed{\texttt{BEGIN V3 := V2; V2 := V1 END}}(\{\texttt{V1}\cdot x,\ \texttt{V2}\cdot y,\ \texttt{V3}\cdot z,\ ...\}) = \{\texttt{V1}\cdot x,\ \texttt{V2}\cdot x,\ \texttt{V3}\cdot y,\ ...\},$$

so the concurrent assignment of

```
BEGIN
  V3 := V2;
  V2 := V1
END
```

is given in the commented version:

```
BEGIN {V2,V3 := V1,V2}
  V3 := V2;
  V2 := V1
END
```

The advantage of concurrent assignments over part meanings is that only changed variables figure in a concurrent assignment. Other variables may be present in the execution state, but if they do not appear on the left side of a concurrent assignment, their values do not change.

Concurrent assignment comments make the effect of a program part explicit. The above examples are straightforward, but consider the following BEGIN statements:

E	*F*
BEGIN	BEGIN
V1 := V2;	V1 := V2;
V2 := V3;	V3 := V1;
V3 := V1	V2 := V3
END	END

in which the second and third assignments are interchanged. We can work out the part functions for E and F, letting x, y, z stand for any characters, as

$$\begin{aligned}
&\boxed{E}(\{\texttt{V1}\cdot x,\ \texttt{V2}\cdot y,\ \texttt{V3}\cdot z,\ ...\}) \\
&= (\boxed{\texttt{V1 := V2}} \circ \boxed{\texttt{V2 := V3}} \circ \boxed{\texttt{V3 := V1}})(\{\texttt{V1}\cdot x,\ \texttt{V2}\cdot y,\ \texttt{V3}\cdot z,\ ...\}) \\
&= (\boxed{\texttt{V2 := V3}} \circ \boxed{\texttt{V3 := V1}})(\{\texttt{V1}\cdot y,\ \texttt{V2}\cdot y,\ \texttt{V3}\cdot z,\ ...\}) \\
&= \boxed{\texttt{V3 := V1}}(\{\texttt{V1}\cdot y,\ \texttt{V2}\cdot z,\ \texttt{V3}\cdot z,\ ...\})
\end{aligned}$$

$= \{V1 \cdot y,\ V2 \cdot z,\ V3 \cdot y,\ ...\}.$

and

$$\boxed{F}(\{V1 \cdot x,\ V2 \cdot y,\ V3 \cdot z,\ ...\})$$
$$= (\boxed{V1\ :=\ V2} \circ \boxed{V3\ :=\ V1} \circ \boxed{V2\ :=\ V3})(\{V1 \cdot x,\ V2 \cdot y,\ V3 \cdot z,\ ...\})$$
$$= (\boxed{V3\ :=\ V1} \circ \boxed{V2\ :=\ V3})(\{V1 \cdot y,\ V2 \cdot y,\ V3 \cdot z,\ ...\})$$
$$= \boxed{V2\ :=\ V3}(\{V1 \cdot y,\ V2 \cdot y,\ V3 \cdot y,\ ...\})$$
$$= \{V1 \cdot y,\ V2 \cdot y,\ V3 \cdot y,\ ...\}.$$

In E, the net effect is to destroy the value of V1 and exchange the values of V2 and V3, as summarized in the concurrent assignment:

```
V1,V2,V3 := V2,V3,V2
```

while F has quite a different effect, namely to give all the variables the original value of V2:

```
V1,V3 := V2,V2
```

Adding concurrent-assignment comments to the statements makes their actions clear:

E

```
BEGIN {V1,V2,V3 := V2,V3,V2}
  V1 := V2;
  V2 := V3;
  V3 := V1
END
```

F

```
BEGIN {V1,V3 := V2,V2}
  V1 := V2;
  V3 := V1;
  V2 := V3;
END
```

E is an important BEGIN statement that exchanges the values of V2 and V3 using V1 as a temporary holding variable; F is often created by mistake when trying to write E.

6.3.2 Symbolic Execution and Trace Tables

The concurrent assignment of a BEGIN statement can be determined by direct inspection or from the part function. However, for complex BEGIN statements, there is a need for a quick, accurate way to obtain the concurrent assignment directly. For example,

```
BEGIN
  V1 := V2;
  V2 := V4;
  V3 := V1;
  V4 := V3
END
```

is tedious to analyze using the definition of part meaning, and direct inspection is likely to get the wrong answer.

Symbolic execution is a method of tracing the values of variables through execution using only their names, not particular values. A *trace table* is a systematic method for carrying out symbolic execution. Like an execution table, a trace table has a row for each statement that is executed, and a column for each variable that might acquire a new value during execution. However, the "values" in a trace table are expressions, and the rows keep track of current expressions in terms of the original, starting expressions. For example, the trace table for the four-assignment BEGIN statement above is:

Statement	V1	V2	V3	V4
V1 := V2	V2	V2	V3	V4
V2 := V4	V2	V4	V3	V4
V3 := V1	V2	V4	V2	V4
V4 := V3	V2	V4	V2	V2

The expressions in the columns represent the *original* values of the named variable. Thus before the first assignment is executed, the variables have their own original values. But following the first assignment (first row in the table), V1 has acquired the value that V2 originally had, and this is shown by listing V2 in the V1 column. The other variables are unchanged, so their values are copied from above. In the second row, V2 acquires the value of V4 (which happens to be its original value). In the third row, the V2 in column V3 comes from the V1 column in the second row, because V3 acquires the current value of V1, which happens to be the original value of V2. The rule is that only *original* values are used in the table, but the proper ones are always available in the row above.

The concurrent assignment for a BEGIN statement can be written directly from a trace table for its constituent statements, by using the list of identifiers at the top of the table on the left side of the assignment and the list from the last row on the right side. The concurrent assignment for the example above is:

```
V1,V2,V3,V4 := V2,V4,V2,V2
```

It simplifies a trace table to omit values that remain unchanged from the row above, making entries only when there is a change. For example, the incorrect attempt to interchange V2 and V3 above has the trace table:

Statement	V1	V2	V3
V1 := V2	V2		
V3 := V1			V2
V2 := V3		V2	

with concurrent assignment

```
V1,V2,V3 := V2,V2,V2
```

or equivalently:

```
V1,V3 := V2,V2
```

The trace table for BEGIN statement

```
BEGIN
  V1 := V4;
  V2 := V3;
  V3 := V2;
  V4 := V1
END
```

is:

Statement	V1	V2	V3	V4
V1 := V4	V4			
V2 := V3		V3		
V3 := V2			V3	
V4 := V1				V4

The concurrent assignment can then document the BEGIN statement:

```
BEGIN {V1,V2 := V4,V3}
  V1 := V4;
  V2 := V3;
  V3 := V2;
  V4 := V1
END
```

This concurrent assignment may be a surprise, because the last two assignments are unnecessary. Chances are that the programmer intended something else, and working out the concurrent assignment has shown up a mistake. If this concurrent assignment was intended, then the BEGIN statement can be simplified by eliminating the last two assignments.

6.3.3 Nested BEGIN Statements

A BEGIN statement may itself be made up of other BEGIN statements. Each constituent statement may alter several variables, and its concurrent assignment can be worked out, just as above. For example,

```
BEGIN
  BEGIN {V1,V3 := V2,V4}
    V1 := V2;
    V3 := V4
  END;
  BEGIN {V2,V3 := V4,V1}
    V2 := V4;
    V3 := V1
  END;
  BEGIN {V1,V4 := V2,V3}
    V1 := V2;
    V4 := V3
  END
END
```

has the inner concurrent assignments already worked out and documented as comments. Then a trace table can be constructed using these concurrent assignments, instead of the original assignments they summarize:

Statement	V1	V2	V3	V4
V1,V3 := V2,V4	V2		V4	
V2,V3 := V4,V1		V4	V2	
V1,V4 := V2,V3	V4			V2

The resulting concurrent assignment,

```
V1,V2,V3,V4 := V4,V4,V2,V2
```

may also be a surprise, because it can be attained more simply with:

```
BEGIN {V1,V2,V3,V4 := V4,V4,V2,V2}
  V1 := V4;
  V3 := V2;
  V2 := V1;
  V4 := V3
END
```

Just to be certain, here is the trace table:

Statement	V1	V2	V3	V4
V1 := V4	V4			
V3 := V2			V2	
V2 := V1		V4		
V4 := V3				V2

6.3.4 Exercises

6.3.1 For each BEGIN statement below, give the concurrent assignment, and demonstrate that it is correct by constructing a trace table.

a)
```
BEGIN
    W1 := W2;
    W2 := W3;
    W3 := W4
END
```

b)
```
BEGIN
    W1 := W2;
    W2 := W3;
    W3 := W4
END
```

c)
```
BEGIN
    W1 := 'A';
    W2 := 'B';
    W3 := 'C';
    BEGIN
      W1 := W3;
      W2 := W1
    END
END
```

d)
```
BEGIN
    Temp := W1;
    W1 := W2;
    W2 := Temp
END
```

6.3.2 Explain how a BEGIN statement can involve k variables on the left sides of its assignments, but its description using a concurrent assignment involves less than k variables on the left.

6.3.3 Is there a CF Pascal BEGIN statement containing two assignment statements that is correctly described by the comment:

```
{V1,V2 := V2,V1}
```

Explain.

6.3.4 Write an actual Pascal assignment statement described by the concurrent assignment

```
< > := < >
```

6.3.5 The statement

```
READ(V1)
```

is in some sense an "assignment" to V1. Explain why it isn't very satisfactory to describe the effect of this "assignment" with a concurrent assignment.

6.3.6

a) Write a fragment of Pascal program that might be described by the concurrent assignment `<V1, V2> := <?, ?>` and place it as an appropriate comment.

b) Explain why concurrent assignments using "?" are never *needed* to describe any actual Pascal assignment statements.

6.3.7

a) Write Pascal statements that are described by the concurrent assignment:

```
V1,V2 := 'A','A'
```

b) Is the answer to a) unique?

c) Is the concurrent assignment `V1,V2 := 'A',V1` different than a)? Explain.

6.4 The Design of Sequential Statements

Preview

Design rules describe the systematic implementation of concurrent assignments in CF Pascal.

New ideas: design rules.

With a simple intended concurrent assignment, it is usually possible to puzzle out a BEGIN statement to implement it, and then to verify that the implementation is or is not correct. However, in complicated cases it is better to design BEGIN statements in a more systematic way than by trial and error.

The symbolic execution of a BEGIN statement provides a basis for design. Observe that every row of a trace table represents the concurrent assignment achieved by the statements down to that row. For example, in the trace table:

Statement	V1	V2	V3
V1 := V2	V2	V2	V3
V2 := V3	V2	V3	V3
V3 := V1	V2	V3	V2

three concurrent assignments can be identified,

```
V1,V2,V3 := V2,V2,V3
V1,V2,V3 := V2,V3,V3
V1,V2,V3 := V2,V3,V2
```

corresponding to the three rows of the table.

In any trace table row, the history of assignments leading up to that row is immaterial to future concurrent assignments; only the information summarized in that row matters. In particular, if the original variable values have been lost, no future assignments can use them. For example, at row 2 of the example above, the original value of V1 has been lost, and the original value of V2 has been destroyed but a copy exists in V1. Any assignments after row 2 (say in row 3) can use only the original values of V2 and V3, because only those expressions exist in row 2 of the trace table. If it was desired to assign the original value of V2 to V3 in row 3, the assignment

```
V3 := V2
```

would be incorrect, since in row 2 V2 is found in the V1 column; therefore, the assignment

```
V3 := V1
```

is required instead.

A design process that proceeds one row of the trace table at a time might be blocked for want of variables to hold values that must be saved; in this case temporary variables must be created. For example, suppose it is desired to exchange the values of V2 and V3, and a first line of the trace

table

Trace table to achieve V2,V3 := V3,V2

Statement	V2	V3
?		

is to be constructed. It cannot be either

```
V2 := V3
```

or

```
V3 := V2
```

because either would destroy a value needed for the final assignment. Therefore, a temporary variable must be created to save a value for later use. A temporary variable is just another variable, with its own column in the trace table. Since it has no final intended value, a temporary can be used as needed to meet another design objective.

6.4.1 A Design Rule for BEGIN statements

Design of a BEGIN statement begins with an objective, stated as an *intended concurrent assignment.* The design rule provides a systematic, step-by-step way to create a BEGIN statement that achieves the intended concurrent assignment.

The design rule will first be explained in an example. Suppose that the intended concurrent assignment is:

```
V1,V2,V3 := 'A',V1,V2
```

For an initial execution state s, the set of values needed to accomplish the assignment is

$H = \{ \boxed{\texttt{'A'}}(s), \boxed{\texttt{V1}}(s), \boxed{\texttt{V2}}(s) \}$.

Care must be taken not to destroy values in this set. Begin by assigning one of the variables its required value, but only if this assignment destroys no value in H. The possible assignments are

```
V1 := 'A'        V2 := V1        V3 := V2
```

but observe that the first two destroy values in H. Therefore, the only possible choice is the third assignment, leading to the partial trace table:

Trace table to achieve V1,V2,V3 := 'A',V1,V2

Statement	V1	V2	V3
V3 := V2	V1	V2	V2

The value $\boxed{\texttt{V2}}(s)$ is now preserved in V3, so the assignment to V2 can be made, with a new partial trace table:

Trace table to achieve V1,V2,V3 := 'A',V1,V2

Statement	V1	V2	V3
V3 := V2	V1	V2	V2
V2 := V1	V1	V1	V2

At this point $\boxed{\texttt{V1}}(s)$ is preserved in V2, so the assignment to V1 can be made, with a final trace table:

Trace table to achieve V1,V2,V3 := 'A',V1,V2

Statement	V1	V2	V3
V3 := V2	V1	V2	V2
V2 := V1	V1	V1	V2
V1 := 'A'	'A'	V1	V2

The last row of this trace table gives the intended concurrent assignment exactly, so the design is completed, and the BEGIN statement can be written and documented as follows:

```
BEGIN {V1,V2,V3 := 'A',V1,V2}
  V3 := V2;
  V2 := V1;
  V1 := 'A'
END
```

If any doubt arises during design, a trace table can be developed simultaneously; however, the table is seldom necessary.

An intended concurrent assignment can be realized as a BEGIN statement using the following design rules:

1. All original values on the right side of the intended concurrent assignment must be preserved at each step (*preservation* rule).
2. Stop when all variables on the left side of the intended concurrent assignment have been assigned their intended values (*stopping* rule).
3. At each step, if a variable can be assigned its intended value, do so; otherwise introduce a temporary variable and assign to it a value that must be preserved (*assignment* rule).

When the design is complete, if temporary variables were required, they should be added to the intended concurrent assignment for documentation.

Example 1: For the intended concurrent assignment

```
Ch2,Ch3,Ch4 := Ch1,Ch2,Ch3
```

the design rules lead to the following steps:

1. `BEGIN {Ch2,Ch3,Ch4 := Ch1,Ch2,Ch3}`

(statement of the design problem);

2.
```
BEGIN {Ch2,Ch3,Ch4 := Ch1,Ch2,Ch3}
  Ch4 := Ch3
```

(only assignment possible—the value of Ch4 need not be preserved);

3.
```
BEGIN {Ch2,Ch3,Ch4 := Ch1,Ch2,Ch3}
  Ch4 := Ch3;
  Ch3 := Ch2
```

(only assignment possible—the value of Ch3 has been preserved in Ch4);

4.
```
BEGIN {Ch2,Ch3,Ch4 := Ch1,Ch2,Ch3}
  Ch4 := Ch3;
  Ch3 := Ch2;
  Ch2 := Ch1
```

(only assignment possible—the value of Ch2 has been preserved in Ch3);

5.
```
BEGIN {Ch2,Ch3,Ch4 := Ch1,Ch2,Ch3}
  Ch4 := Ch3;
  Ch3 := Ch2;
  Ch2 := Ch1
END
```

(all variables have been assigned their intended values).

In this case the design rules force a unique sequence of assignments.

Example 2: A temporary variable will be needed to realize the intended concurrent assignment:

```
Ch1,Ch2 := Ch2,Ch1
```

The design rules lead to the following steps:

1.
```
BEGIN {Ch1,Ch2 := Ch2,Ch1}
```

2.
```
BEGIN {Ch1,Ch2 := Ch2,Ch1}
    Temp1 := Ch2
```

(neither Ch1 nor Ch2 can be assigned without violating the preservation rule, so preserve the value of Ch2 in Temp1);

3.
```
BEGIN {Ch1,Ch2 := Ch2,Ch1}
    Temp1 := Ch2;
    Ch2 := Ch1
```

(only assignment possible—the value of Ch2 has been preserved in Temp1);

4.
```
BEGIN {Ch1,Ch2 := Ch2,Ch1}
    Temp1 := Ch2;
    Ch2 := Ch1;
    Ch1 := Temp1
```

(only assignment possible—the value of Ch1 has been preserved in Ch2);

5.
```
BEGIN {Ch1,Ch2 := Ch2,Ch1}
    Temp1 := Ch2;
    Ch2 := Ch1;
    Ch1 := Temp1
```

(all variables have been assigned their intended values);

6.
```
BEGIN {Ch1,Ch2,Temp1 := Ch2,Ch1,Ch2}
    Temp1 := Ch2;
    Ch2 := Ch1;
    Ch1 := Temp1
  END
```

(Temp1 has been added to the concurrent assignment for documentation).

In step 2, the choice to preserve the value of Ch2 is arbitrary. If instead Ch1 were preserved, the result would be:

```
BEGIN {Ch1,Ch2,Temp1 := Ch2,Ch1,Ch1}
  Temp1 := Ch1;
  Ch1 := Ch2;
  Ch2 := Temp1
END
```

As assignments are created according to the design rules, the portion of the design remaining can be isolated as a new problem.

Example 3: For an intended concurrent assignment

```
Ch1,Ch2,Ch3,Ch4 := Ch4,Ch3,Ch2,Ch2
```

the first assignment is necessarily to Ch1 by the preservation rule, giving:

1.
```
BEGIN {Ch1,Ch2,Ch3,Ch4 := Ch4,Ch3,Ch2,Ch2}
    Ch1 := Ch4
```

At this point, a new design problem can be deduced as a concurrent assignment, namely

```
2. BEGIN {Ch1,Ch2,Ch3,Ch4 := Ch4,Ch3,Ch2,Ch2}
     Ch1 := Ch4;
     BEGIN {Ch2,Ch3,Ch4 := Ch3,Ch2,Ch2}
```

and only Ch4 can be assigned:

```
3. BEGIN {Ch1,Ch2,Ch3,Ch4 := Ch4,Ch3,Ch2,Ch2}
     Ch1 := Ch4;
     BEGIN {Ch2,Ch3,Ch4 := Ch3,Ch2,Ch2}
       Ch4 := Ch2;
       BEGIN {Ch2,Ch3 := Ch3,Ch2}
```

The remaining design problem is exactly the one solved in Example 2, so the design can be completed by copying that result:

```
4. BEGIN {Ch1,Ch2,Ch3,Ch4,Temp := Ch4,Ch3,Ch2,Ch2,Ch3}
     Ch1 := Ch4;
     BEGIN {Ch2,Ch3,Ch4 := Ch3,Ch2,Ch2}
       Ch4 := Ch2;
       BEGIN {Ch2,Ch3 := Ch3,Ch2}
         Temp := Ch3;
         Ch3 := Ch2;
         Ch2 := Temp
       END
     END
   END
```

6.4.2 Exercises

6.4.1 Check the final design in Example 3 above using a trace table.

6.4.2 Design BEGIN statements to implement the following intended assignments:

a) `V1,V2,V3 := V3,V2,V1`

b) `V1,V2,V3,V4 := V4,V3,V2,V1`

c) `V1,V2,V3,V4 := V3,V3,V1,V1`

d) `V1,V2 := 'C','B'`

6.4.3 Find a different application of the design rules to Example 3 that will eliminate the use of a temporary variable, or explain why this is impossible.

6.4.4 What happens when the design rules are applied to the concurrent assignment:

```
< > := < >
```

Is the result correct?

6.4.5 What is the minimum number of assignment statements required to accomplish any concurrent assignment that has four variables on the left side? What is the maximum number that might be required? Explain.

6.4.6

a) Give an example of an intended concurrent assignment and its realization using the design rules that uses more than one temporary variable.

b) The assignment design rule could be modified to free temporary variables as soon as possible:

   At each step, if a variable can be assigned its intended value, do so using a value stored in a temporary variable if possible; otherwise introduce or reuse a temporary variable and assign a value to be preserved to it (*optimized assignment* rule).

If the optimized rule is used, is a single temporary variable sufficient for the design of any BEGIN statement? Explain.

6.5 Chapter Summary

The semantics of sequential CF Pascal programs can be defined in a straightforward manner. Since CF Pascal assignment statements are limited to having a right side which is either a variable declared as a CHAR or a literal character, the semantics are kept simple. The null statement does nothing and therefore has an identity function as its meaning. BEGIN statements have as meaning the composed meaning of their constituent statements. WRITE statements append to the value of the appropriate file. READ statements set a value taken from a file. A procedure statement has as its meaning, the meaning of the BEGIN statement from its declaration.

The concurrent assignment is a notation for expressing the meaning of a BEGIN statement more concisely than as a part function. BEGIN statements can be analyzed directly using trace tables, yielding the concurrent-assignment meaning.

The design rules for BEGIN statements provide a systematic method for realizing a given concurrent assignment.

CHAPTER 7

THE CALCULUS OF CONDITIONAL STATEMENTS

Chapter Preview

Chapter 7 continues the development of the program calculus for conditional statements and culminates in a design rule for creating correct conditional statements.

The execution table for a program containing no iteration cannot have more rows than the program has statements. When there are IF statements, the input data of the table may cause different parts of the program to be used, but the program parts listed will be a subset of the program statements. IF statements have meanings that can be composed to form the meaning of the program, but the meaning of the parts, and hence of the whole, will depend on the conditions in those statements.

7.1 The Meaning of Conditional Statements

Preview

IF statement meaning depends on the meaning of the <condition> that is tested in statement execution. The <condition> is a Boolean expression whose meaning can be defined just as character-expression meanings were. The meaning of an IF statement can then be given in terms of its constituents, which are the <condition> and the statements executed depending on the condition's outcome.

Much of the power (and complication) in computer programs comes from conditional statements, which provide a means for making decisions based not only on program input, but on intermediate values determined during execution of the program. In isolation, however, a single IF statement is not complicated, and its meaning can be easily defined.

7.1.1 Meaning of Boolean Expressions

Since conditional statements include Boolean expressions as program parts, the box notation is extended to cover Boolean expressions. Just as the meaning of a character expression is a mapping from execution states to character values, so the meaning of a Boolean expression maps from execution states to Boolean values, that is, to the set {TRUE, FALSE} described in Chapter 3. For the purpose of giving the semantics of CF Pascal using only character values, the Boolean values TRUE and FALSE will be represented by the characters **T** and **F** respectively, here and in Chapter 8. Because Boolean expressions may be combined to create arbitrarily complex <condition>s using the operators AND, OR, and NOT with parentheses, the definition of meaning must be recursive.

The simplest <condition> is either two character expressions with a <comparison> operator, or one of the file conditions EOF or EOLN. These meanings must be given first.

Let f be a file identifier, and x and y strings. Define:

$$\boxed{\texttt{EOF (} f \texttt{)}} = \{\langle s,\ \mathbf{T}\rangle : s(f) = \langle x, \dagger\dagger,\ \mathbf{R}\rangle\}$$
$$\cup \{\langle s,\ \mathbf{F}\rangle : s(f) = \langle x, y,\ \mathbf{R}\rangle,\ y \neq \dagger\dagger\},$$

$$\boxed{\texttt{EOLN (} f \texttt{)}} = \{\langle s,\ \mathbf{T}\rangle : s(f) = \langle x, y,\ \mathbf{R}\rangle,\ \Theta\ y = /\}$$
$$\cup \{\langle s,\ \mathbf{F}\rangle : s(f) = \langle x, y,\ \mathbf{R}\rangle,\ y \neq \dagger\dagger,\ \Theta\ y \neq /\}.$$

When there is no identifier f, the conditions apply to INPUT, and the meaning can be found by substituting INPUT for f in the definitions:

$\boxed{\texttt{EOF}} = \boxed{\texttt{EOF (INPUT)}}$

$\boxed{\texttt{EOLN}} = \boxed{\texttt{EOLN (INPUT)}}$

(Note that when EOF has the value **T**, the string y in EOLN is empty, so neither condition applies and EOLN is undefined.)

For example, to determine the value of (say) $\boxed{\texttt{EOF (F1)}}(s)$ in a state s containing the pair abbreviated F1·<**A2**, **345/**, **R**>, one of the two sets whose union is the definition of $\boxed{\texttt{EOF (F1)}}$ must be identified by the condition following the colon. In this case only the second set fits, because $y = \dagger\mathbf{345/}\dagger \neq \dagger\dagger$. Thus **F** is paired with this s, that is,

$$\boxed{\texttt{EOF (F1)}}(s) = \mathbf{F}.$$

Three of the character-comparison operations are defined as follows:

$$\boxed{c = d} = \{<s,\ \mathbf{T}>: \boxed{c}(s) \text{ is the same character as } \boxed{d}(s)\}$$
$$\cup \{<s,\ \mathbf{F}>: \boxed{c}(s) \text{ is not the same character as } \boxed{d}(s)\},$$

$$\boxed{c < d} = \{<s,\ \mathbf{T}>: \boxed{c}(s) \text{ precedes } \boxed{d}(s)\}$$
$$\cup \{<s,\ \mathbf{F}>: \boxed{c}(s) \text{ does not precede } \boxed{d}(s)\},$$

$$\boxed{c > d} = \boxed{d < c}.$$

For example, let s be a state containing Ch·**A**, so $\boxed{\texttt{Ch}}(s) = \mathbf{A}$. Then

$$\boxed{\texttt{Ch = 'A'}}(s) = \mathbf{T},$$
$$\boxed{\texttt{Ch = 'X'}}(s) = \mathbf{F},$$
$$\boxed{\texttt{Ch < 'A'}}(s) = \mathbf{F},$$
$$\boxed{\texttt{Ch > 'A'}}(s) = \mathbf{F}.$$

In the definition of $\boxed{c < d}$ the result turns on whether the first character value precedes the second. This idea is defined for all Pascal machines, for all possible characters recognized by the machine. However, there is no standard definition of "precedes" except that no character precedes itself, and:

A precedes **B** precedes **C** ... precedes **Z**

a precedes **b** precedes **c** ... precedes **z**

0 precedes **1** precedes **2** ... precedes **9**.

The relationships between other characters are defined, but vary in different Pascal machines. In examples, this text will use only the cases above, such as **0** precedes **5**.

On this basis, the composite Boolean <condition>s can be given meanings. Let b be a Boolean condition whose meaning is known. Then

$$\boxed{\texttt{NOT (}\ b\ \texttt{)}} = \{<s,\ \mathbf{T}>: \boxed{b}(s) = \mathbf{F}\} \cup \{<s,\ \mathbf{F}>: \boxed{b}(s) = \mathbf{T}\}.$$

Similarly, for Boolean conditions b and c, define:

$$\boxed{b\ \texttt{AND}\ c} = \{<s,\ \mathbf{T}>: \boxed{b}(s) = \mathbf{T}, \boxed{c}(s) = \mathbf{T}\} \cup$$
$$\{<s,\ \mathbf{F}>: \boxed{b}(s) = \mathbf{T}, \boxed{c}(s) = \mathbf{F}\} \cup$$
$$\{<s,\ \mathbf{F}>: \boxed{b}(s) = \mathbf{F}, \boxed{c}(s) = \mathbf{T}\} \cup$$
$$\{<s,\ \mathbf{F}>: \boxed{b}(s) = \mathbf{F}, \boxed{c}(s) = \mathbf{F}\},$$

$$\boxed{b\ \texttt{OR}\ c} = \{<s,\ \mathbf{T}>: \boxed{b}(s) = \mathbf{T}, \boxed{c}(s) = \mathbf{T}\} \cup$$
$$\{<s,\ \mathbf{T}>: \boxed{b}(s) = \mathbf{T}, \boxed{c}(s) = \mathbf{F}\} \cup$$
$$\{<s,\ \mathbf{T}>: \boxed{b}(s) = \mathbf{F}, \boxed{c}(s) = \mathbf{T}\} \cup$$
$$\{<s,\ \mathbf{F}>: \boxed{b}(s) = \mathbf{F}, \boxed{c}(s) = \mathbf{F}\}.$$

The Boolean operations NOT, AND, and OR of Chapter 3 are mirrored in these definitions, if **T** is taken to be TRUE, and **F** to be FALSE. Then the definitions can be given as follows:

$\boxed{\texttt{NOT } b}(s) = \text{NOT } \boxed{b}(s)$,

$\boxed{b \texttt{ AND } c}(s) = \boxed{b}(s) \text{ AND } \boxed{c}(s)$,

$\boxed{b \texttt{ OR } c}(s) = \boxed{b}(s) \text{ OR } \boxed{c}(s)$.

The Pascal operators NOT, AND, OR appear on the left side of these definitions. On the right side, only the component conditions b and c appear, along with the (mathematical) Boolean operations NOT, AND, and OR. The Pascal meaning is thus defined to agree with the mathematical meaning, and this is very convenient in calculations, where all the facts about Boolean expression from Chapter 3 can be brought to bear.

Here are some examples of Boolean-condition meanings:

$\boxed{\texttt{NOT (Ch = 'A')}}(\{\texttt{Ch·}\mathbf{A}, \ldots\}) = \mathbf{F}$,

$\boxed{\texttt{(Ch1 = 'A') AND (Ch2 <> Ch1)}}(\{\texttt{Ch1·}\mathbf{A}, \texttt{ Ch2·}\mathbf{F}, \ldots\}) = \mathbf{T}$,

$\boxed{\texttt{(Ch1 = 'A') OR (Ch2 <> Ch1)}}(\{\texttt{Ch1·}\mathbf{B}, \texttt{ Ch2·}\mathbf{F}, \ldots\}) = \mathbf{T}$.

The remaining part meanings for Boolean conditions can be defined in terms of those already given. Let c and d be character expressions. Define:

$\boxed{c \texttt{ <> } d} = \boxed{\texttt{NOT(} c \texttt{ = } d \texttt{)}}$,

$\boxed{c \texttt{ <= } d} = \boxed{\texttt{NOT(} c \texttt{ > } d \texttt{)}}$,

$\boxed{c \texttt{ >= } d} = \boxed{\texttt{NOT(} c \texttt{ < } d \texttt{)}}$.

To work out a meaning for a composite Boolean condition it is necessary to apply the definitions in the proper order so that at each step the result is carried nearer to one of the base meanings, which eventually can be evaluated. For example, let

$s = \{\texttt{Ch·}\mathbf{F}, \texttt{ INPUT·<}\mathbf{Line}\square, \texttt{ }\mathbf{34}/, \texttt{ }\mathbf{R}>, \ldots\}$.

Then

$\boxed{\texttt{(Ch <> '\#') AND (NOT(EOLN))}}(s)$

$= \boxed{\texttt{Ch <> '\#'}}(s) \text{ AND } \boxed{\texttt{NOT(EOLN)}}(s)$

$= \boxed{\texttt{NOT(Ch = '\#')}}(s) \text{ AND } \boxed{\texttt{NOT(EOLN)}}(s)$

$= (\text{NOT } \boxed{\texttt{Ch = '\#'}}(s)) \text{ AND } (\text{NOT } \boxed{\texttt{EOLN}}(s))$

$= (\text{NOT } \mathbf{F}) \text{ AND } (\text{NOT } \mathbf{F}) = \mathbf{T} \text{ AND } \mathbf{T} = \mathbf{T}$.

This example could probably be correctly evaluated by inspection, but recourse to the formal definition may be required to handle more difficult cases or to record reasoning about them.

Part meanings now have the following possible domains and ranges:

Table of Domains and Ranges of Part Meanings

Program part	*Domain*	*Range*
program	lists of strings	lists of strings
program header	lists of strings	execution states
final period	execution states	lists of strings
block	execution states	execution states
declaration	execution states	execution states
statement	execution states	execution states
character expression	execution states	characters
Boolean expression	execution states	Boolean values

7.1.2 Meaning of IF Statements

The part meaning of an IF statement can now be expressed simply and directly as a function. As for other statements, the meaning is a mapping from execution states to execution states. Let b be a Boolean <condition> and let T and E be statements. Define:

$$\boxed{\texttt{IF } b \texttt{ THEN } T \texttt{ ELSE } E} =$$
$$\{<s, \boxed{T}(s)>: \boxed{b}(s) = \mathbf{T}\} \cup \{<s, \boxed{E}(s)>: \boxed{b}(s) = \mathbf{F}\}.$$

The first set contains all state pairs for which condition b holds of argument s, and the second set contains those pairs in which b does not hold of s. Since no value can be both **T** and **F**, there are no states s that occur as argument in both sets. It is important to note the way that "undefined" can occur in the definition. The sets are not evaluated in any particular order; they simply contain or fail to contain certain pairs. If $\boxed{b}$ fails to be defined for some s, neither condition holds, so s is not an argument in either set; that is, the whole part meaning is undefined for an s on which $\boxed{b}$ is undefined. Even if $\boxed{b}$ is defined at s, the part meaning may be undefined, if $\boxed{b}(s) = \mathbf{T}$ and $\boxed{T}$ is undefined at s, or if $\boxed{b}(s) = \mathbf{F}$ and $\boxed{E}$ is undefined at s.

For example, let u be:

```
IF V1 < V2
THEN
  V1 := V2
ELSE
  V2 := V1
```

Then

$$\boxed{u} = \{<s, \boxed{\texttt{V1 := V2}}(s)>: \boxed{\texttt{V1 < V2}}(s) = \mathbf{T}\} \cup$$
$$\{<s, \boxed{\texttt{V2 := V1}}(s)>: \boxed{\texttt{V1 < V2}}(s) = \mathbf{F}\}.$$

For state $s = \{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot\mathbf{B}, ...\}$,

$$\boxed{\texttt{V1 < V2}}(\{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot\mathbf{B}, ...\}) = \mathbf{T},$$

so the value of $\boxed{u}$ for this argument is given by the value of $\boxed{\texttt{V1 := V2}}$:

$$\boxed{u}(\{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot\mathbf{B}, ...\}) = \boxed{\texttt{V1 := V2}}(\{\texttt{V1}\cdot\mathbf{A},\ \texttt{V2}\cdot\mathbf{B}, ...\}) = \{\texttt{V1}\cdot\mathbf{B},\ \texttt{V2}\cdot\mathbf{B}, ...\}.$$

The definition of IF statement part meaning agrees with the results of an execution table, but the ideas are different. In an execution table there is a definite sequence of activity: (for the example) evaluate the condition V1 < V2, then do the assignment V1 := V2. The function definition on the other hand has no concept of sequence in it.

An IF statement without an ELSE part is composed of just a Boolean condition b and a statement T:

$$\boxed{\texttt{IF } b \texttt{ THEN } T} = \{<s, \boxed{T}(s)>: \boxed{b}(s) = \mathbf{T}\} \cup \{<s, s>: \boxed{b}(s) = \mathbf{F}\}.$$

For example,

$$\boxed{\texttt{IF EOF THEN WRITELN('Oops...')}}$$
$$(\{\texttt{INPUT}\cdot<\mathbf{XXX},\ \mathbf{XX}/,\ \mathbf{R}>,\ \texttt{OUTPUT}\cdot<\dagger\dagger,\ \dagger\dagger,\ \mathbf{W}>, ...\})$$
$$= \{\texttt{INPUT}\cdot<\mathbf{XXX},\ \mathbf{XX}/,\ \mathbf{R}>,\ \texttt{OUTPUT}\cdot<\dagger\dagger,\ \dagger\dagger,\ \mathbf{W}>, ...\}$$

because

$$\boxed{\texttt{EOF}}(\{\texttt{INPUT}\cdot<\mathbf{XXX},\ \mathbf{XX}/,\ \mathbf{R}>,\ \texttt{OUTPUT}\cdot<\dagger\dagger,\ \dagger\dagger,\ \mathbf{W}>, ...\}) = \mathbf{F},$$

and the second set of the definition is the one in which this state is paired with itself.

7.1.3 Exercises

7.1.1 Give the value of each of the following, showing each step in the evaluation from the definitions:

a) [V1 = V2]({V2·3, V1·X, ...})

b) [V1 <> V2]({V2·3, V1·X, ...})

c) [EOLN(Int)]({Int·<ZZ/, /ZZ/, R>, ...})

d) [(Ch <= '9') OR (Ch >= '6')]({Ch·3, ...})

7.1.2 In each part below a <condition> c and a state s are given. If the meaning $[c]$ is undefined on s, explain why; if it is defined, give the value. Show all the calculations.

a) Ch = 'X', {Ch·B, ...}

b) EOF, {INPUT·<††, ††, R>, ...}

c) EOLN, {INPUT·<††, ††, R>, ...}

d) EOF(Who), {Who·<123, ††, W>, ...}

7.1.3 Let C be the IF statement:

```
IF V1 < V2
THEN
  V1 := 'X'
ELSE
  V1 := V2
```

Evaluate $[C]$ on each of the following states:

a) {V1·X, V2·Y, ...}

b) {V1·#, V2·#, ...}

c) {V1·7, V2·5, ...}

7.1.4 In each part below, a state s and an IF statement Q are given. If $[Q]$ is not defined on s, explain why; otherwise calculate $[Q](s)$. Show all the intermediate steps in either case.

a) {INPUT·<††, A/, R>, Ch·?, ...}

```
IF NOT(EOF)
THEN
  READ(Ch)
ELSE
  Ch := 'Z'
```

b) {INPUT·<†1†, /2/3/, R>, Ch·?, ...}

```
IF EOLN
THEN
  READ(Ch)
```

c) {Ch·X, INPUT·<†/†, ††, R>, Temp·<††, /More/, R>, ...}

```
IF EOLN(Temp)
THEN
  READ(Ch)
```

d) {Temp·<No/, ††, R>, OUTPUT·<Yes□or□, ††, W>, ...}

```
IF EOLN(Temp)
THEN
  WRITELN('Done')
ELSE
  WRITELN
```

7.1.5

a) Explain why

$$\{<s, \boxed{T}(s)>: \boxed{b}(s) = \mathbf{T}\}$$

is *not* the meaning for

IF b THEN T

b) Give a Pascal statement for which this *is* the meaning.

7.2 The Analysis of Conditional Statements

Preview

The meaning of an IF statement can be summarized by a conditional assignment, an extension of the idea of a concurrent assignment. The symbolic execution of programs with conditional statements can be recorded in a set of trace tables, with a new column added to keep track of condition values. Conditional trace tables can be used to derive the meaning of an IF statement as a conditional assignment, or to analyze conditional statements directly.

7.2.1 Conditional Assignments

A *conditional assignment* is a generalization of a concurrent assignment that allows IF statements to be analyzed directly, just as the concurrent assignment made BEGIN-statement analysis easier. The part meaning of an IF statement has two possibilities, depending on the <condition>. The conditional assignment displays the alternatives along with the Boolean condition that decides between them. The syntax is not part of Pascal, but can be attached to statements as a comment.

Conditional assignments can be defined recursively by the following rules:

1. A concurrent assignment is a conditional assignment.
2. If b is a Boolean condition and c is a conditional assignment, then ($b \rightarrow c$) is a conditional assignment.
3. If b is a Boolean condition and c, d are conditional assignments, then ($b \rightarrow c$) | d is a conditional assignment.

A conditional assignment that happens to be a concurrent assignment C can be written ($\mathbf{T} \rightarrow C$). Then a conditional assignment is of the form

$$(b_1 \rightarrow c_1) \mid (b_2 \rightarrow c_2) \mid \ldots \mid (b_n \rightarrow c_n)$$

for any number of Boolean conditions (b_1, b_2, ..., b_n) and conditional assignments (c_1, c_2, ..., c_n). In a state s, its meaning is the first conditional assignment, say c_i, such that

(i) all Boolean conditions before b_i (i.e., b_1, b_2, ..., b_{i-1}) have value **F** in state s, and

(ii) Boolean condition b_i has value **T**.

The meaning is undefined for state s if any of the following:

(i) none of the Boolean conditions b_1, ..., b_n evaluates to **T** in s;

(ii) the first Boolean expression that does not evaluate to **F** is undefined for s;

(iii) the conditional assignment selected by the Boolean expression is undefined for s.

For example, the IF statement

```
IF V1 <= V2
THEN
  V1 := '2'
ELSE
  BEGIN
    V1 := V2;
    V2 := 'A'
  END
```

has a conditional assignment:

```
(V1 <= V2 → V1 := '2') | (T → V1,V2 := V2,'A')
```

expressing the knowledge that for an initial execution state s, should `V1 <= V2`(s) be **T**, the result is described by the concurrent assignment

```
V1 := '2'
```

while if `V1 <= V2`(s) is **F**,

```
V1,V2 := V2,'A'
```

describes the result, because its Boolean condition is **T**. On the other hand,

```
(NOT(EOLN) → Ch := 'Z') | (EOF → Ch := 'A' )
```

describes a statement that aborts for the execution state

{INPUT·<**All□gone/**, ††, **R**>, ...}

because `EOLN` is not defined on this state so the conditional assignment is not defined.

The Boolean conditions are called *guards* of the assignments; in the general form each term consists of a *guarded assignment*. In the special case that a guard always holds, a term may be written without it, for example:

```
((V1 = V3) OR (V1 <> V3)) → Ch1,V3 := 'A',V1)
```

is the same as the concurrent assignment:

```
(Ch1,V3 := 'A',V1)
```

When such an unguarded term appears, there is no point in including terms following it: its condition *is* **T**, so the meaning must be fixed when that term is reached, if not before. The form first introduced to describe an IF statement, for example, shown as a comment on:

```
{(V1 <= V2 → V1 := '2') | (V1,V2 := V2,'A')}
IF V1 <= V2
THEN
  V1 := '2'
ELSE
  BEGIN
    V1 := V2;
    V2 := 'A'
  END
```

is thus one with an unguarded last term. It could as well have been written

```
(V1 <= V2 → V1 := '2') | (V1 > V2 → V1,V2 := V2,'A')
```

since if the first guard evaluates to **F**, the second must evaluate to **T**.

The concurrent assignment

```
< > := < >
```

in which no variable values are changed, with a **T** guard,

```
(T → < > := < >)
```

can be written simply

```
( )
```

and expresses the meaning of a null statement: an identity function. Including this term at the end of a conditional assignment guarantees that a meaning will always be defined: if all the guards before prove **F**, the meaning will be identity from the last term.

An IF statement without an ELSE part can be described by a conditional assignment, as shown by the following example:

```
{(V1 < V2 → V1 := V2) | ( )}
IF V1 < V2
THEN
  V1 := V2
```

The final term, a null concurrent assignment, is crucial because when the condition is **F**, the IF statement does have the meaning of an identity function. The conditional assignment

```
(V1 < V2 → V1 := V2)
```

is *not* the meaning of this statement, because it states that when the value of V1 does not precede that of V2, the IF statement aborts, which is not the case.

Consider a conditional assignment in which the guarded assignment is another conditional assignment; e.g.:

```
(Ch > 'A' → ((Ch = 'B' → Ch := 'C') |
        (Ch = 'C' → Ch := 'D')))
```

In order for the inner conditional expression to be the meaning, its outer guard must evaluate to **T**, and for either of the inner assignments to be used *their* guards must evaluate to **T**. Therefore the outer guard can be distributed among the inner guards with AND operators to produce a conditional assignment with one less nesting level. For this example, the following conditional assignment describes the same meaning:

```
((Ch > 'A') AND (Ch = 'B') → Ch := 'C') |
((Ch > 'A') AND (Ch = 'C') → Ch := 'D')
```

Conditional assignments within conditional assignments capture the meaning of nested IF statements. For example, consider:

```
IF V1 < V2
THEN
  V1 := V2
ELSE
  IF V2 < V1
  THEN
    V2 := V1
```

A conditional assignment can be written directly:

```
(V1 < V2 → V1 := V2) | ((V2 < V1 → V2 := V1) | ( ))
```

which is an abbreviation for:

```
(V1 < V2 → V1 := V2) | (T → ((V2 < V1 → V2 := V1) | ( )))
```

Distributing the final condition yields the three terms:

```
(V1 < V2 → V1 := V2) | (V2 < V1 → V2 := V1) | ( )
```

It can be seen that in this meaning V1 and V2 each acquire the larger (in the sense of the "precedes" ordering) of their original values. This could be described by inventing an operator max for "maximum value of" in the concurrent assignment:

```
V1,V2 := max(V1,V2),max(V1,V2)
```

Thus a good comment for the original statement would be:

```
{V1,V2 := max(V1,V2),max(V1,V2)}
IF V1 < V2
THEN
  V1 := V2
ELSE
  IF V2 < V1
  THEN
    V2 := V1
```

This IF statement was probably written to be efficient, to use no unnecessary assignments, and thus was hard to understand at a glance. But the concurrent assignment tells the story. And if this fragment of program appeared in a BEGIN statement, the concurrent assignment could be used in working out its trace table.

The comment

```
{V1,V2 := max(V1,V2),max(V1,V2)}
```

is more concise and revealing than the conditional assignment

```
{(V1 < V2 → V1 := V2) | (V2 < V1 → V2 := V1) | ( )}
```

that expresses the same meaning. Any well understood mathematical function can be used for a revealing comment in this way. For example, to describe a program part that sorts a file `F1` back into itself,

```
{F1 := sort(F1)}
```

is concise and meaningful if `sort(F1)` has a mathematical definition as a function from file values (lists of strings) to file values.

As an illustration of a more general comment, consider the IF statement:

```
IF Digit = '0' THEN Digit := '1' ELSE
IF Digit = '1' THEN Digit := '2' ELSE
IF Digit = '2' THEN Digit := '3' ELSE
IF Digit = '3' THEN Digit := '4' ELSE
IF Digit = '4' THEN Digit := '5' ELSE
IF Digit = '5' THEN Digit := '6' ELSE
IF Digit = '6' THEN Digit := '7' ELSE
IF Digit = '7' THEN Digit := '8' ELSE
IF Digit = '8' THEN Digit := '9'
```

whose meaning could be described by the comment:

```
{((Digit >= '0') AND (Digit < '9') → Digit := character to the
     right of Digit in the string 0123456789)
  | ( )}
```

The mathematical function here is "character to the right of `Digit` in the string †0123456789†."

7.2.2 Conditional Trace Tables

The trace tables that deal with BEGIN statements can be extended to handle IF statements. A *conditional trace table* is a trace table with an additional column of conditions, namely those required for the assignments in the table to take place. For example, the BEGIN statement R

```
BEGIN
  V1 := V2;
  V2 := V3;
  IF V1 < V2
  THEN
    V3 := V1
  ELSE
    V3 := V2
END
```

will use one of two sequences of assignments, namely

```
V1 := V2; V2 := V3; V3 := V1
```

or

```
V1 := V2; V2 := V3; V3 := V2
```

depending on whether the THEN or ELSE part of the IF statement is selected during execution. Each sequence can be handled by a separate conditional trace table labeled R(T), R(E) (T and E for THEN or ELSE part) as follows:

Trace Table for R(T)

Statement	Condition	V1	V2	V3
V1 := V2		V2		
V2 := V3			V3	
IF V1 < V2	V2 < V3			
THEN V3 := V1				V2

Each row of the table shows values in terms of the original variables. The condition V2 < V3 in the third row is the value of V1 < V2, because at that point V1 has the original value of V2, and V2 has the original value of V3 (obtained from the row above). The net result for this conditional trace table is a conditional assignment

```
(V2 < V3 → V1,V2,V3 := V2,V3,V2)
```

where the guard is the final condition of the trace table, and the concurrent assignment is obtained as before.

Trace Table for R(E)

Statement	Condition	V1	V2	V3
V1 := V2		V2		
V2 := V3			V3	
IF V1 < V2	V2 >= V3			
ELSE V3 := V2				V3

In this case, the condition is that required to reach the ELSE statement, and the net result is the conditional assignment

```
(V2 >= V3 → V1,V2 := V2,V3)
```

The conditional assignment for the entire BEGIN statement R is obtained by combining these two conditional assignments:

```
(V2 < V3 → V1,V2,V3 := V2,V3,V2) | (V2 >= V3 → V1,V2 := V2,V3)
```

As another example, BEGIN statement S has two IF statements:

```
BEGIN
  IF V1 < V2
  THEN
    V1 := V2
  ELSE
    V2 := V3;
  V1 := V3;
  IF V2 < V3
  THEN
    V2 := V3
  ELSE
    V3 := V1
END
```

which will execute in one of four sequences, depending on which of the THEN or ELSE parts of the two IF statements are selected. Identify these sequences using T or E for the THEN or ELSE alternatives in order, so S(T,T) means the two THEN parts are taken, while S(E,T) means that the first ELSE is taken, then the second THEN. Each sequence has a conditional trace table:

Trace Table for S(T,T)

Statement	Condition	V1	V2	V3
IF V1 < V2	V1 < V2			
THEN V1 := V2		V2		
V1 := V3		V3		
IF V2 < V3	V2 < V3			
THEN V2 := V3			V3	

For this particular sequence to be executed, the two conditions must both hold. Thus, the guard for the assignments in this table is

```
(V1 < V2) AND (V2 < V3)
```

which makes the conditional assignment for S(T,T):

```
((V1 < V2) AND (V2 < V3) → V1,V2 := V3,V3)
```

Similarly,

Trace Table for S(T,E)

Statement	Condition	V1	V2	V3
IF V1 < V2	V1 < V2			
THEN V1 := V2		V2		
V1 := V3		V3		
IF V2 < V3	V2 >= V3			
ELSE V3 := V1				V3

with conditional assignment:

```
((V1 < V2) AND (V2 >= V3) → V1 := V3)
```

The table

Trace Table for S(E,T)

Statement	Condition	V1	V2	V3
IF V1 < V2	V1 >= V2			
ELSE V2 := V3			V3	
V1 := V3		V3		
IF V2 < V3	V3 < V3			
THEN V2 := V3				

yields:

```
((V1 >= V2) AND (V3 < V3) → V1,V2 := V3,V3)
```

The guard is always **F**, because of V3 < V3, so the execution sequence represented by S(E,T) cannot occur.

The table

Trace Table for S(E,E)

Statement	Condition	V1	V2	V3
IF V1 < V2	V1 >= V2			
ELSE V2 := V3			V3	
V1 := V3		V3		
IF V2 < V3	V3 >= V3			
ELSE V3 := V1				V3

yields:

```
(V1 >= V2 → V1,V2 := V3,V3)
```

Collecting the three possible conditional assignments S(T,T), S(T,E), S(E,E) produces the conditional assignment for S, namely

```
((V1 < V2) AND (V2 < V3) → V1,V2 := V3,V3) |
((V1 < V2) AND (V2 >= V3) → V1 := V3) |
(V1 >= V2 → V1,V2 := V3,V3)
```

This meaning can be simplified using Boolean logic. Notice that the concurrent assignments in the first and third terms are the same. That is, the same actions should be taken if either of these guards is **T**. Because the second guard can never be **T** if either of the others is, the order of the guarded assignments can be changed, and the first and third combined:

```
(((V1 < V2) AND (V2 < V3)) OR (V1 >= V2) → V1,V2 := V3,V3) |
((V1 < V2) AND (V2 >= V3) → V1 := V3)
```

Distributing the OR (from Boolean identities 10 and 8, Section 3.2.5), the first guard is equivalent to

```
((V1 < V2) OR (V1 >= V2)) AND ((V2 < V3) OR (V1 >= V2))
```

which by the character ordering with OR is the same as

```
T AND ((V2 < V3) OR (V1 >= V2))
```

so the conditional assignment could be replaced by:

```
((V2 < V3) OR (V1 >= V2) → V1,V2 := V3,V3) |
((V1 < V2) AND (V2 >= V3) → V1 := V3)
```

The second guard is always **T** when the first is **F**, so this is the same meaning as:

```
((V2 < V3) OR (V1 >= V2) → V1,V2 := V3,V3) | (V1 := V3)
```

The assignment to V1 is the same under both conditions, so this meaning goes with the program part:

```
BEGIN
  IF ((V2 < V3) OR (V1 >= V2))
  THEN
    V2 := V3;
  V1 := V3
END
```

which is a considerable simplification of the original IF statement, developed a step at a time through analysis.

7.2.3 Comparing Conditional Assignments

Two IF statements may have the same meaning, although they appear quite different. As a simple example,

```
IF V1 < V2 THEN V1 := V2 ELSE V2 := V1
  =IF V2 < V1 THEN V2 := V1 ELSE V1 := V2
```

because both have the meaning of the concurrent assignment:

```
V1,V2 := max(V1,V2),max(V1,V2)
```

The execution of these statements is different, and they have different descriptions as conditional assignments:

```
(V1 < V2 → V1 := V2) | (V1 >= V2 → V2 := V1)
```

and

```
(V2 < V1 → V2 := V2) | (V2 >= V1 → V1 := V2)
```

respectively. These meanings can be compared by reducing both to a common conditional assignment. The last terms can be broken into two; for example, the first above becomes

```
(V1 < V2 → V1 := V2) | (V1 = V2 → ) | (V1 > V2 → V2 := V1)
```

because in the equality case the assignment is superfluous. The expansion of the second conditional assignment above gives exactly the same terms, but in a different order. Because the guards are mutually exclusive, the order is arbitrary, so these two conditional assignments express the same meaning, another way of proving that the IF statements they describe mean the same thing.

For more complicated comparisons between conditional assignments, systematic procedures are needed to determine if they describe the same meaning.

Recall the definition of the form of a conditional assignment:

1. A concurrent assignment is a conditional assignment.
2. If b is a Boolean condition and c is a conditional assignment, then $(b \rightarrow c)$ is a conditional assignment.
3. If b is a Boolean condition and c, d are conditional assignments, then $(b \rightarrow c) \mid d$ is a conditional assignment.

Rules 2 and 3 may lead to nested guards, and the first step in comparing two expressions is to eliminate these and obtain a *reduced form* in which each term has at most one guard. This can be done by repeatedly distributing outer guards into inner ones until the nesting is eliminated. For example, in

```
(V1 = V2 → (V2 < V3 → V1 := V3)) | (V3 := V2)
```

the first term becomes

```
((V1 = V2) AND (V2 < V3) → V1 := V3)
```

and one distribution suffices. When all nested guards have been eliminated, a conditional assignment must be in one of three forms:

1. (c)

2. $(\ b \rightarrow c\ )$

3. $(\ b \rightarrow c\ )\ \mid d$

where b is a Boolean expression, and d is a conditional assignment, but now c is a concurrent assignment.

Suppose that two conditional assignments in reduced form are to be compared to see if they express the same meaning. Each might be in any of the three forms, so the comparison might fall into any of the slots in the table:

Form of first assignment	*Form of second assignment*		
	(g)	$(f \rightarrow g)$	$(f \rightarrow g) \mid h$
(c)	case 1	case 2	case 3
$(b \rightarrow c)$		case 4	case 5
$(b \rightarrow c) \mid d$			case 6

The blank entries are symmetric, so there are only six cases to be treated.

Case 1. The two conditional assignments to be compared are both concurrent assignments. Two concurrent assignments express the same meaning if and only if the variables in the list on the left are the same, and the corresponding expressions on the right are also the same. (Variables whose values do not change are omitted from the assignments.) For example, the conditional assignment

```
(V1,V2,V3 := V3,V1,'A')
```

does not express the same meaning as

```
(V3,V1 := 'A',V3)
```

because although the variables that appear do correspond to the same expressions, `V2` does not appear in both.

Case 2. The first conditional assignment is a concurrent assignment (c), and the second is a guarded concurrent assignment $(f \rightarrow g)$. These express the same meaning if and only the guard f always holds (that is, $\boxed{f}$ is defined on every state s, and $\boxed{f}(s) =$ `T`), and the concurrent assignments c and g have the same meaning (Case 1 above).

Case 3. The first conditional assignment is (c), and the second is $(f \rightarrow g) \mid h$. These can be put into the same form by artificially introducing the guard into the first without changing its meaning,

$(f \rightarrow c) \mid c$

which converts the problem to Case 6 below.

Case 4. The first conditional assignment is $(b \rightarrow c)$, and the second is $(f \rightarrow g)$. Each of these is undefined except for states on which its guard takes the value `T`. Hence for the conditional assignments to have the same meaning requires that $\boxed{b} = \boxed{f}$. In states s for which $\boxed{b}(s) = \boxed{f}(s) =$ `T`, the concurrent assignments c and g must agree.

For example, the assignment

```
(V1 = V2 → V1,V2,V3 := V3,V1,'A')
```

has the same meaning as

```
(V1 = V2 → V3,V1 := 'A',V3)
```

because only states in which $\boxed{\texttt{V1 = V2}}$ takes the value `T` are involved, and on these states the assignment to `V2` adds nothing to the first. Similarly, the meaning of

```
(V1 = 'X' → V2 := 'X')
```

is the same as that of

```
(V1 = 'X' → V2 := V1)
```

because these are both defined only when `V1` has the value `X`, and when that occurs, the meanings are both to give `V2` the value `X`.

Case 5. The first conditional assignment is $(b \rightarrow c)$, and the second is $(f \rightarrow g) \mid h$; where c and g are concurrent assignments, but h is a conditional assignment. The second assignment is broken into two cases in its two terms; these can be inserted into the first assignment by rewriting it as

$$(f \rightarrow (b \rightarrow c)) \mid (b \rightarrow c)$$

or

$$(f \text{ AND } b \rightarrow c) \mid (b \rightarrow c)$$

Thus for these conditional assignments to agree, three subdomains are of interest:

On states where	*The condition is*
f and b both hold	c and g must agree
b holds but f doesn't	$(b \rightarrow c)$ and h must agree
b doesn't hold	$(f \rightarrow g) \mid h$ must be undefined

The first subdomain reduces to Case 1 above. The second subdomain is an example of Case 2 or Case 4 above, or a simpler example of Case 5, depending on the form of h.

Case 6. The first conditional assignment is $(b \rightarrow c) \mid d$, and the second is $(f \rightarrow g) \mid h$, where c and g are concurrent assignments, but d and h are conditional assignments. Inserting each assignment's condition in the other, these are:

$$(f \text{ AND } b \rightarrow c) \mid (b \rightarrow c) \mid d$$

and

$$(b \text{ AND } f \rightarrow g) \mid (f \rightarrow g) \mid h$$

The following table describes the situations under which these assignments have the same meaning:

On states where	*The condition is*	*Reduced to*
f and b both hold	c and g must agree	Case 1
f holds but b doesn't	d and g must agree	Case 1, 2, or 3
b holds but f doesn't	c and h must agree	Case 1, 2, or 3
neither f nor b holds	d and h must agree	Cases 1-6

In each of the Cases 1-6, this analysis will either decide if the two conditional assignments have the same meaning, or will reduce the situation to one of the cases for further analysis, with the assignments to be further analyzed being simpler than the originals. Thus eventually the decision will come down to examples of Case 1 or Case 4 for direct comparison.

For example, the conditional assignment b:

```
(V1 = V2 → V1 := V3) | (V1 = V3 → V2 := V1) | (V1,V2 := V3,V1)
```

can be shown to have the same meaning as c:

```
(V1,V2 := V3,V1)
```

as an example of Case 3. Inserting the first condition from b into c

```
(V1 = V2 → V1,V2 := V3,V1) | (V1,V2 := V3,V1)
```

gives an instance of Case 6. For states in which the value of `V1 = V2` is **T**, the meanings of c and `V1 := V3` must be the same (Case 4), and they are, since in this situation the assignment to `V2` in c is unnecessary. On the other hand, where `V1 = V2` is **F**, it is required that c and

```
(V1 = V3 → V2 := V1) | (V1,V2 := V3,V1)
```

be the same. This is an example of Case 3 again, reduced to Case 6 by inserting the condition in c:

```
(V1 = V3 → V1,V2 := V3,V1) | (V1,V2 := V3,V1)
```

Now the states to consider are those in which `V1 = V3` is **T**, where c must be the same as `V2 := V1`, as Case 4 shows that it is, since the assignment to `V1` is redundant. When `V1 = V3` is **F**, it is required that c be the same as the final term of b, which is c itself, an example of Case 1. Thus the

two original conditional assignments have the same meaning.

7.2.4 Exercises

7.2.1 In each part below, determine the conditional assignment for the given IF statement.

a)
```
IF V1 < V2
THEN
  V1 := V2
```

b)
```
IF V2 < V3
  THEN
    V1 := V3
  ELSE
    V1 := V2
```

c)
```
IF (V1 < V2) OR (V2 < V1)
  THEN
    V1 := V2
  ELSE
    V2 := V1
```

7.2.2 Show that the nested conditional assignment

```
(V1 < V2 → V1 := V2) |
  (V1 >= V2 → ((V2 < V1 → V2 := V1) | (V2 >= V1 → )))
```

simplifies to:

```
(V1 < V2 → V1 := V2) | (V2 < V1 → V2 := V1) | (V1 = V2 → )
```

What program do these give a meaning for?

7.2.3 In each part below, give a conditional assignment for the given IF statement, and simplify the assignment.

a)
```
IF V1 < V2
  THEN
    V3 := V1
  ELSE
    IF V2 < V3
    THEN
      V2 := V1
```

b)
```
IF (V1 < V2) OR (V2 < V3)
  THEN
    V1 := V2
  ELSE
    V2 := V3
```

c)
```
IF V1 = V2
  THEN
    V3 := V1
  ELSE
    IF V2 = V3
    THEN
      V1 := V3
```

d)
```
BEGIN
   V1 := V4;
   V3 := V2;
   IF V1 < V3
   THEN
     V2 := V1
   ELSE
     V4 := V3
END
```

7.2.4 Consider the conditional assignment:

(V1 = 'Q' → V1 := 'V') | (V1 = 'V' →)

Give a Pascal statement with this meaning.

7.2.5 Using the max function from Section 7.2.1, write a concurrent assignment with the same meaning as the conditional assignment:

(V1 < V2 → V1 := V2) | (V1 >= V2 →)

7.2.6 Explain why

```
IF b
THEN
  A
ELSE
  C
```

is a statement whose meaning is

(*b* → *A*) | (NOT(*b*) → *C*)

but that

```
IF b
THEN
  A
```

is *not* a statement with the meaning:

(*b* → *A*)

Can you give a program that has the latter meaning?

7.2.7 Suppose *b* and *c* are Boolean expressions that are never true together, but both may be false together.

a) Explain why

```
IF b
THEN
  X
ELSE
  IF c
  THEN
    Y
```

is *not* a program with the meaning:

(*b* → *X*) | (*c* → *Y*)

b) Give a program that does have this meaning.

c) Give a conditional assignment that is the meaning of the given IF statement.

7.2.8 Use conditional trace tables to find the conditional assignments that describe each statement:

a)
```
BEGIN
    V1 := V2;
    V2 := V3;
    IF V1 < V3
    THEN
      V2 := V1
    ELSE
      V3 := V1
END
```

b)
```
IF V1 < V2
THEN
  BEGIN
    V2 := V3;
    IF V1 < V3
    THEN
      V2 := V1
  END
```

c)
```
BEGIN
    IF V1 < V2
    THEN
      V3 := V1
    ELSE
      V3 := V2;
    IF V2 < V3
    THEN
      V1 := V2
    ELSE
      V1 := V3
END
```

d)
```
BEGIN
    V1 := V2;
    V2 := V3;
    IF (V1 < V2) OR (V2 < V3)
    THEN
      BEGIN
        V2 := V3;
        V3 := V1
      END
END
```

7.2.9 In a conditional trace table, suppose that the composite condition of the table simplifies to **F**. What use is then made of this table, and why?

7.2.10 Suppose a BEGIN statement has been analyzed so that a conditional assignment is available for each of its statements. Describe a method for generating all the conditional trace tables for this statement.

7.2.11 Compare each pair of conditional assignments, and either prove them to have the same meaning, or find a state on which their meanings disagree:

a)
```
(V1 <= V2 → V1 := V2)
(V2 <= V1 → V2 := V1)
```

b) (V1 = V2 → V1 := V2)
(V2 = V1 → V2 := V1)

c) (V1 <> V2 → V1 := V2)
(V2 <> V1 → V2 := V1)

7.2.12 Compare each pair of conditional assignments, and either prove them to have the same meaning, or find a state on which their meanings disagree:

a) (V1 < V2 → V3 := V1) | (V1 = V2 → V3 := V2)
| (V1 > V2 → V3 := V2)
(V1 > V2 → V3 := V2) | (V1 := V3)

b) (V1 = V2) → V1 := V3) | (V1 := V2)
(V1 <> V2 → V1 := V2) | (V1 := V3)

c) (V2 < V1 → V3 := V2) | (V2 >= V1 → V3 := V1)
(V2 = V1 → V3 := V1) | (V2 <> V1 → V3 := V2)

7.3 The Design of Conditional Statements

Preview

An IF statement can be systematically designed to have the meaning of a given conditional assignment.

The design of IF statements permits an intended assignment to be split into two simpler assignments. The creative part of the design is the choice of a condition to make the split.

7.3.1 Design Rule for IF statements

Consider the intended concurrent assignment:

```
V1 := max(V2,V3)
```

This meaning can be attained by comparing V2 and V3, and using two assignments:

```
IF V2 > V3
THEN
  V1 := V2
ELSE
  V1 := V3
```

The choice of Boolean expression is critical. For example, any of

```
V2 < V3      V2 <= V3      V2 >= V3
```

will do as well (with different assignments to follow), but none of

```
V1 < V2      V2 = V3      V2 <> V3
```

is any good at all. The critical idea, which is at the heart of creative design, is to find a Boolean expression for the IF condition such that both the THEN statement and the ELSE statement are easier to complete than the intended assignment. These statements *can* be easier to complete, because the knowledge that the Boolean expression takes **T** or **F** values can be exploited within them.

For any Boolean expression b and any statement S, if

$$\text{domain}(\boxed{S}) \subseteq \text{domain}(\boxed{b})$$

then

$\boxed{S} = \boxed{\texttt{IF } b \texttt{ THEN } S \texttt{ ELSE } S}$.

That is, any statement can be converted into an IF statement in a trivial way by choosing a Boolean expression that is defined for all states where the statement executes normally. This trivial IF statement does not have THEN and ELSE parts simpler than the original statement, but it could happen that

$\boxed{S} = \boxed{\texttt{IF } b \texttt{ THEN } T \texttt{ ELSE } E}$

for some statements T and E that are less complex than S, because each is used only for some cases, separated by b. When S is not a CF Pascal statement, but only an intended concurrent or conditional assignment, an IF statement is often the best choice for its implementation. In the example above, the concurrent assignment

```
V1 := max(V2,V3)
```

was realized with two simple assignment statements, based on the separate cases of an IF statement.

The design rule that results from this analysis, though simpler than for BEGIN statements, puts more burden on the creative powers of the designer. Given an intended assignment A, construct the IF statement

```
IF b
THEN {b → A}
  T
ELSE {NOT b → A}
  E
```

for some b such that T and E are simpler than A. The IF statement will have the intended meaning if the domain of A is a subset of domain($\boxed{b}$) and the conditional-assignment comments correctly describe T and E.

Example 1: For the intended assignment:

```
V1,V2 := max(V1,V2),max(V1,V2)
```

choose a splitting condition of `V1 < V2`; then the design becomes:

```
{V1,V2 := max(V1,V2),max(V1,V2)}
IF V1 < V2
THEN {(V1 < V2 → V1,V2 := V2,V2)}
  V1 := V2
ELSE {(V1 >= V2 → V1,V2 := V1,V1)}
  V2 := V1
```

Example 2: For the intended assignment:

```
V1,V2 := max(V1,V2),min(V1,V2)
```

choose a splitting condition of `V1 < V2`; then the design becomes:

```
{V1,V2 := max(V1,V2),min(V1,V2)}
IF V1 < V2
THEN {(V1 < V2 → V1,V2 := V2,V1)}
  {use BEGIN statement}
ELSE {(V1 >= V2 →  )}
  {no action needed}
```

This can be completed as follows:

```
{V1,V2 := max(V1,V2),min(V1,V2)}
IF V1 < V2
THEN
  BEGIN {V1,V2 := V2,V1}
    Temp := V1;
    V1 := V2;
    V2 := Temp
  END
```

Example 8: For the intended statement:

```
V1,V2 := max(V1,V2),V1
```

a design is:

```
{V1,V2 := max(V1,V2),V1}
IF V1 < V2
THEN {(V1 < V2 → V1,V2 := V2,V1)}
  {use BEGIN statement}
ELSE {(V1 >= V2 → V1,V2 := V1,V1)}
  V2 := V1
```

7.3.2 Designing IF Statements for Conditional Assignments

Because conditional assignments express the meaning of IF statements directly, an intended conditional assignment can be immediately translated into a design. It is best to put the conditional assignment in reduced form (eliminate nested guards by distributing the outer ones into the inner ones). In that way there is less chance of confusing the nesting in the designed IF statement. The most general form the intended assignment can take is then

$$(b \dot{\rightarrow} C) \mid D$$

where b is a Boolean condition, C is a concurrent assignment, and D is a conditional assignment. The design is:

```
IF b
THEN {b → C}
  {BEGIN statement design for C}
ELSE
  {continue the design for D}
```

The design for D will be done in the same way, leading to a regular program structure in which each item of the intended assignment has its IF statement, nested from left to right in the conditional assignment.

For example, the following is a design for the intended assignment that is its heading comment:

```
{((Ch <= '9') AND (Ch >= '0') → Result := 'D')
  | (Ch <> 'N' → Result := Ch) | (Ch,Result := 'Y','N')}
IF (Ch <= '9') AND (Ch >= '0')
THEN {((Ch <= '9') AND (Ch >= '0') → Result := 'D')}
  Result := 'D'
ELSE
  IF Ch <> 'N'
  THEN {(Ch <> 'N' → Result := Ch)}
    Result := Ch
  ELSE
    BEGIN {Ch,Result := 'Y','N'}
      Ch := 'Y';
      Result := 'N'
    END
```

7.3.3 Exercises

7.3.1 In each part below, design an IF statement that implements the intended assignment.

a) `V1,V2 := max(V1,V2),min(V2,V3)`

b) `V1,V2,V3 := max(V1,V2,V3),min(V1,V2,V3),min(V1,V2,V3)`

c) `V1,V2 := max('0',V2),min('9',V1)`

7.3.2 In each part below, design an IF statement that implements the intended assignment.

a) `(V2 < V1 → V3 := V2) | (V2 >= V1 → V3 := V1)`

b) `(V2 = V1 → V3 := V1) | (V2 <> V1 → V3 := V2)`

7.3.3 Design an IF statement for the conditional assignment with nested guards:

```
(V1 <> V2 →
   (V1 < V3 → V1,V2 := V2,V1) | (V3 < V2 → V1,V2 := 'A','Z'))
     | (V2 = V3 → V1 := V2)
```

a) directly from the assignment;

b) by first distributing the guard to get the assignment in reduced form.

Use comments to make the structure of each design clear.

7.4 Chapter Summary

Like character expressions, Boolean expressions in CF Pascal have meanings that are mappings from execution states to values, Boolean values **T** and **F**. IF statements have meanings that map states into states, but these meanings separate into two sets: the pairs of states for which the Boolean expression takes the value **T** on the argument, and those for which it takes the value **F**. In the first set the THEN part of the IF statement determines the mapping; in the second set it is the ELSE part.

Conditional assignments provide a concise notation for describing the meaning of IF statements, by expressing the Boolean condition as in CF Pascal, and the alternative actions as concurrent or conditional assignments. The conditional trace tables provide a means of direct analysis and for obtaining the conditional assignment of a given IF statement. It is possible to systematically compare two conditional assignments to see if they express the same meaning.

Design of IF statements requires more creativity than design of BEGIN statements. The designer must choose a Boolean expression for the IF statement that splits the intended assignment into simpler parts, and this can only be done through experience and trial. Once the splitting expression is chosen, however, the knowledge of being in the THEN or ELSE part simplifies the design of statements that must be placed there. If the intended assignment is a conditional one, of course, the design follows its form in a mechanical way.

CHAPTER 8

THE CALCULUS OF ITERATION STATEMENTS

Chapter Preview

The program calculus is extended to cover WHILE statements. Although there is no general method for calculating part functions of WHILE statements, and even deciding termination is in general an unsolvable problem, methods are given for checking the correctness of iteration statements, and for designing them.

The meanings of programs without iteration can be calculated directly, a step at a time. First, the meaning of expressions is obtained; then meanings for assignment statements, READ and WRITE statements; then for BEGIN and IF statements; then for declarations, blocks, and the program header and period. In every case, the part meaning is defined using the text of a program part, and that text is used only once in the definition. For iteration, the texts of the WHILE condition and DO statement can be used many times in execution, so the meaning definition is more involved.

There is no general way to calculate the meaning of an arbitrary iteration statement directly (any more than there is a way to trisect an angle with ruler and compass). But there is a general way to check the correctness of any candidate meaning for an iteration statement. Since design by stepwise refinement provides candidate meanings before statements are written to satisfy them, checking these meanings ensures the correctness of iteration statements inserted in a design.

8.1 The Meaning of Iteration Statements

Preview

The execution of a WHILE statement can be thought of as repeating the entire statement, leading to a recursive definition of meaning. This meaning can also be described in terms of the number of times the DO part is repeated.

8.1.1 The Meaning of WHILE Statements Defined Recursively

The meaning of a WHILE statement can be defined directly from the way it is executed. Let w be

```
WHILE b DO d
```

where b is a Boolean expression and d is a statement. In an execution state where b has the value **T**, d is executed, perhaps changing the state, and then w is executed again with the new state. If b has the value **F** for some state, then w acts as a null statement on that state. This behavior of w can be described by the IF statement

```
IF b THEN BEGIN d; w END
```

and thus the formal definition is:

$$\boxed{w} = \boxed{\texttt{IF } b \texttt{ THEN BEGIN } d\texttt{; } w \texttt{ END}}.$$

Simple WHILE statements can be analyzed directly. For example let v be:

```
WHILE V1 < V2 DO V1 := V2
```

By definition:

$$\boxed{v} = \boxed{\texttt{IF V1 < V2 THEN BEGIN V1 := V2; } v \texttt{ END}}.$$

Since the v in the right side will always be applied to a state on which the assignment `V1 := V2` has just acted, the Boolean expression `V1 < V2` in this v will always evaluate to **F**. Therefore v

can be removed from the right side, because it has the effect of a null statement. Thus

$$[\![v]\!] = [\![\texttt{IF V1 < V2 THEN V1 := V2}]\!].$$

The meaning of v is now easy to understand, since the iteration has been eliminated.

As a second example let u be

```
WHILE V1 <= V2 DO V1 := V2
```

which differs from v only in the comparison operator of the condition. This small change leads to radically different behavior. By definition,

$$[\![u]\!] = [\![\texttt{IF V1 <= V2 THEN BEGIN V1 := V2; } u \texttt{ END}]\!]$$

As for v, the state that the repeated WHILE statement sees has the same value for V1 and V2, because of the assignment. But the Boolean expression of u is always **T** for such a state, so the WHILE statement will be used again, and again the expression will be **T**, and so on forever. That is, for any state in which the Boolean condition is **T**, an endless execution will ensue. No such argument states can enter $[\![u]\!]$; and for any other state, $[\![u]\!]$ is the identity. Thus,

$$[\![u]\!] = \{<s, s>: [\![\texttt{V1 <= V2}]\!](s) = \mathbf{F}\}.$$

In terms of a conditional assignment, this meaning is:

```
(V1 > V2 → )
```

As a more complex example, let t be:

```
WHILE V1 < V2
DO
  BEGIN
    V1 := V2;
    V2 := V3
  END
```

The DO part of t can be summarized in the concurrent assignment:

```
V1,V2 := V2,V3
```

If the DO part were repeatedly executed, its trace table would look like:

Statement	V1	V2
V1,V2 := V2,V3	V2	V3
V1,V2 := V2,V3	V3	V3
V1,V2 := V2,V3	V3	V3
...		

That is, after the second iteration, both V1 and V2 would hold the original value of V3 and from then on nothing would change. This trace table provides an easy way to analyze t. The condition for termination is V1 >= V2, and the table shows that this always occurs after no more than two iterations. The second iteration occurs only when the original value of V2 does not precede that of V3 (because those are the values for V1 and V2, respectively, following the first iteration). This information is summarized in the conditional assignment meaning for t:

```
(V1 >= V2 → ) | (V2 >= V3 → V1,V2 := V2,V3)
    | (V1,V2 := V3,V3)
```

The first term is the case of no iterations, the second of one iteration, and the third of two iterations. Writing out the full guard for each term:

```
(V1 >= V2 → ) | (V1 < V2 → (V2 >= V3 → V1,V2 := V2,V3))
   | (V1 < V2 → (V2 < V3 → V1,V2 := V3,V3))
```

and distributing into reduced form:

```
(V1 >= V2 → ) | ((V1 < V2) AND (V2 >= V3) → V1,V2 := V2,V3)
   | ((V1 < V2) AND (V2 < V3) → V1,V2 := V3,V3)
```

The three guards of the reduced form define non-intersecting regions in domain($\boxed{t}$). In each region the guarded assignment gives the meaning of t.

8.1.2 Counting Iterations in WHILE Statements

In an execution table, the effect of a WHILE statement arises from repetition of the statement(s) of its DO part. The repetition continues until the termination condition is met. When the number of repetitions can be determined, it provides another way to understand the meaning of a WHILE statement.

Let w be

WHILE b DO d

where b is a Boolean expression and d is a statement, as before. Domain($\boxed{w}$) is the set of execution states for which the execution of w terminates. For each state $s \in$ domain($\boxed{w}$), the iteration count must be some natural number k. (Otherwise an endless execution would occur for state s, and it would not be in domain($\boxed{w}$).) For example, in the third WHILE statement t above, k was 0, 1, or 2. Several facts about w and the initial state s can be deduced from k.

If $k = 0$ then $\boxed{b}(s) =$ **F**, because otherwise there would be at least one iteration. Furthermore, $\boxed{w}(s) = s$ because d is not executed.

If $k = 1$, then

$$\boxed{b}(s) = \mathbf{T},$$
$$\boxed{w}(s) = \boxed{d}(s),$$
$$(\boxed{d} \circ \boxed{b})(s) = \mathbf{F}.$$

The first equation is needed for any iterations to occur. The second reflects the effect of exactly one execution of the DO part d. The third explains why a second iteration does not occur. After the DO part is executed, the new state is $\boxed{d}(s)$, and in this state the Boolean condition must fail, that is

$$\boxed{b}(\boxed{d}(s)) = (\boxed{d} \circ \boxed{b})(s) = \mathbf{F}.$$

If $k = 2$, then

$$\boxed{b}(s) = \mathbf{T},$$
$$(\boxed{d} \circ \boxed{b})(s) = \mathbf{T},$$
$$\boxed{w}(s) = \boxed{d} \circ \boxed{d}(s),$$
$$(\boxed{d} \circ \boxed{d} \circ \boxed{b})(s) = \mathbf{F}.$$

Here the first two evaluations of $\boxed{b}$ must allow the iteration to continue, the DO part d will be used twice, and the third evaluation of $\boxed{b}$ must cause termination.

The pattern of facts is clear, and can be expressed in general as follows. For $k \geq 0$ iterations of w beginning with state $s \in$ domain($\boxed{w}$):

$$(\boxed{d}^0 \circ \boxed{b})(s) = \mathbf{T},$$
$$(\boxed{d}^1 \circ \boxed{b})(s) = \mathbf{T},$$
$$\ldots$$
$$(\boxed{d}^{k-1} \circ \boxed{b})(s) = \mathbf{T},$$
$$\boxed{w}(s) = \boxed{d}^k(s),$$
$$(\boxed{d}^k \circ \boxed{b})(s) = \mathbf{F}.$$

The power notation means repeated composition. For any relation r,

$r^0 = \{<x, x>: x \in \text{domain}(r)\}$,

$r^n = r \circ r^{n-1}$, for $n > 0$.

The special case of zero compositions of r, written r^0, is defined as an identity function. This happens to be just the right definition to handle the case of zero iterations of a WHILE statement.

In the example WHILE statement t above, the facts for two iterations beginning with state s would be written

$\boxed{\texttt{V1 < V2}}(s) = \mathbf{T}$,

$(\boxed{\texttt{V1 := V2; V2 := V1}}^1 \circ \boxed{\texttt{V1 < V2}})(s) = \mathbf{T}$,

$\boxed{t}(s) = \boxed{\texttt{V1 := V2; V2 := V1}}^2(s)$,

$(\boxed{\texttt{V1 := V2; V2 := V1}}^2 \circ \boxed{\texttt{V1 < V2}})(s) = \mathbf{F}$.

The convenience and power of these iteration facts is that in many cases the number of iterations is an effective way to partition domain($\boxed{w}$) for study and analysis. For example, a WHILE statement to flush blanks will iterate exactly as often as the number of leading blanks in the future part of a file.

An iterative definition of WHILE statement meaning can be given along the lines suggested by this discussion:

$\boxed{\texttt{WHILE } b \texttt{ DO } d}$

$= \{<s, t>$: for some natural number k, the following conditions hold:

1. $t = \boxed{d}^k(s)$,
2. for every natural number i less than k, $(\boxed{d}^i \circ \boxed{b})(s) = \mathbf{T}$,
3. $\boxed{b}(t) = \mathbf{F}\}$

This definition can be seen to be the same as the recursive one

$\boxed{\texttt{WHILE } b \texttt{ DO } d} = \boxed{\texttt{IF } b \texttt{ THEN BEGIN } d\texttt{; WHILE } b \texttt{ DO } d \texttt{ END}}$

by considering the cases of each iteration count $k = 0, 1, 2, \ldots$ However, the recursive definition is the one we will use.

A WHILE statement w can be analyzed by considering the termination condition for each $s \in$ domain($\boxed{w}$), to find the iteration count k. The meaning is then a k-fold composition of the meaning of the DO part. For example, consider the WHILE statement:

```
WHILE V1 < V2
DO
  BEGIN
    V1 := V3;
    V3 := V2;
    V2 := V4
  END
```

The DO part has the meaning of the concurrent assignment:

```
V1,V2,V3 := V3,V4,V2
```

Three iterations of this meaning are shown in the trace table:

Statement	V1	V2	V3
V1,V2,V3 := V3,V4,V2	V3	V4	V2
V1,V2,V3 := V3,V4,V2	V2	V4	V4
V1,V2,V3 := V3,V4,V2	V4	V4	V4

From the table it can be seen that the conditions for termination in terms of the original values of the variables are as follows:

Iterations	Condition
0	V1 >= V2
1	V3 >= V4
2	V2 >= V4
3	V4 >= V4

Thus there can be no more than three iterations, because the termination condition is always satisfied after the third. From these two tables the conditional-assignment meaning of the WHILE statement can be constructed. The guards come from the iteration-count table, and the concurrent assignments from the trace table:

```
(V1 >= V2 →  ) | (V3 >= V4 → V1,V2,V3 := V3,V4,V2) |
(V2 >= V4 → V1,V2,V3 := V2,V4,V4) | (V1,V2,V3 := V4,V4,V4)
```

As another illustration, consider the WHILE statement intended to flush blanks from INPUT:

```
WHILE (Ch = ' ') AND (NOT EOF)
DO
  READ(Ch)
```

Suppose the initial execution state for this WHILE statement is

$S = \{$Ch·□, INPUT·$<x$, □□□**THE** & y, **R**$>$, ...$\}$

Where x and y are any character strings. The execution table will contain

After Execution of	Condition	INPUT	Ch
		x & □□□**THE** & y	□
WHILE Ch = ' '	**T**		
READ(Ch)		x & □□□**THE** & y	□
WHILE Ch = ' '	**T**		
READ(Ch)		x & □□□**THE** & y	□
WHILE Ch = ' '	**T**		
READ(Ch)		x & □□□**THE** & y	□
WHILE CH = ' '	**T**		
READ(Ch)		x & □□□**THE** & y	**T**
WHILE Ch = ' '	**F**		

That is, the WHILE statement is iterated four times, and terminates in the fifth evaluation of the WHILE condition. On the other hand, if the initial execution state for this WHILE statement is

$\{$Ch·**N**, INPUT·$<x$, y, **R**$>$, ...$\}$,

then the Boolean expression Ch = ' ' has value **F** at the outset, and the WHILE statement is iterated zero times.

This information is enough to deduce the meaning of this WHILE statement as the conditional assignment:

```
((Ch = ' ') AND (INPUT = <x, y, R>) →
     Ch, INPUT := C, <x & u ∇ C, v, R>) |
 ((Ch <> ' ') OR (INPUT = <x, ††, R>) →  )
```

where

$y = u \nabla C \,\&\, v$, and

u is a (possibly empty) string of blanks,

C is a character other than blank or line marker, and

x, v are arbitrary strings.

Considerable liberties have been taken with the conditional-assignment form, utilizing expressions that are not part of Pascal, but whose meaning is clear.

8.1.3 Exercises

8.1.1 Is it possible to write a WHILE statement *in isolation* that will always, in any context,

a) iterate exactly *once?*

b) *never* iterate at all?

c) never terminate?

d) always iterate more than 1000 times?

Explain your answers and give examples where possible.

8.1.2 In each part, determine a conditional assignment for the WHILE statement by (i) writing a trace table of DO statement iterations; (ii) determining the number of iterations for any condition; and (iii) collecting results of (i) and (ii) into the conditional assignment.

a)
```
WHILE V1 < V2
DO
  V1,V2 := V2,V1
```

b)
```
WHILE V1 <= V2
DO
  V1,V2 := V2,V1
```

c)
```
WHILE V2 < V3
DO
  V2,V3 := V3,V1
```

d)
```
WHILE V3 < V1
DO
  V1,V3 := V2,V1
```

e)
```
WHILE V1 < V2
DO
  V1,V3 := V3,V2
```

f)
```
WHILE V1 < V2
DO
  V1 := V2
```

g)
```
WHILE (V1 < V2) OR (V2 < V3)
DO
  BEGIN
    V1 := V2;
    V2 := V3
  END
```

h)
```
WHILE (V1 < V2) AND (V2 < V3)
DO
  BEGIN
    V2 := V3;
    V1 := V2
  END
```

i)
```
WHILE Ch = ' '
DO
  READ (Ch)
```

j)
```
WHILE (Ch >= '0') AND (Ch <= '9')
DO
  READ (Ch)
```

k)
```
WHILE (Ch >= '0') AND (Ch <= '9')
DO
  BEGIN
    WRITE (FOut,Ch);
    READ (FIn, Ch)
  END
```

8.1.3 Let b be any Boolean expression, and S any statement. For each part, determine the part function of the given WHILE statement.

a)
```
WHILE b
DO
  {statement is empty}
```

b)
```
WHILE Ch = Ch
DO
  S
```

c)
```
WHILE Ch <> Ch
DO
  S
```

8.1.4 In the iterative definition of WHILE-statement meaning, the case $k = 0$ is combined with the other cases. Write out this case from the definition, and show that it agrees with the discussion that lead up to the definition.

8.2 The Analysis of Iteration Statements

Preview

A general method is given to verify that a candidate meaning for a WHILE statement is correct.
New ideas: WHILE statement verification rule.

Because the meaning of a WHILE statement w is defined in terms of w itself, calculating the meaning directly is possible only for very simple examples. In general there is no way to calculate w from the definition. Neither can the iterative definition be used for direct calculation, because the number of iterations cannot in general be determined. However, if a meaning is given, a candidate meaning for a WHILE statement, there *is* a general method for *checking* it. This checking method requires only calculations such as IF- and BEGIN-statement meanings, and composition of meanings. Conditional assignments are a valuable aid in dealing with the necessary calculations.

8.2.1 The WHILE Statement Verification Rule

Let w be any WHILE statement with Boolean expression b and DO statement d:

WHILE b DO d

Let f be a candidate for the meaning $\boxed{w}$, that is, a function one hopes to show the same as $\boxed{\text{WHILE } b \text{ DO } d}$. The WHILE statement verification rule is a general method for deciding if

$$f = \boxed{w}$$

holds or not, and rigorously proving this equality if it does hold.

To use a method that checks a given function rather than calculating one, there must be a source for the candidate meaning. In program development using stepwise refinement, the meaning is given at each step before statements to obtain it are designed. In this case then a candidate

meaning will always be available. When a WHILE loop has been designed in a refinement step, its meaning can be checked against the desired meaning. If they agree, the design is correct.

On the other hand, to calculate the meaning of a WHILE statement that is already written, the intuitive analysis using execution tables, trace tables, and case analysis by iteration counts can be used to obtain a candidate meaning. If this candidate is verified, it can be used to summarize the action of the WHILE statement.

In either case, the verification rule can be used to confirm or deny the equality between the WHILE statement meaning and the candidate meaning. If the equality fails in stepwise refinement, the WHILE statement must be changed. If the equality fails in calculating the meaning of an existing statement, the candidate meaning must be changed.

The plan for developing the WHILE statement verification method is to solve the definition for the statement's meaning, which occurs on both sides of the equation. This requires putting the definition in a slightly different form. Written in full, the definition is:

$$\boxed{\texttt{WHILE } b \texttt{ DO } d} = \boxed{\texttt{IF } b \texttt{ THEN BEGIN } d\texttt{; WHILE } b \texttt{ DO } d \texttt{ END}}.$$

An ELSE part could be added to the IF statement on the right side without altering anything, so long as the ELSE part acts as a null statement. An ELSE part of WHILE *b* DO *d* has this property, because to invoke the ELSE part, *b* would have to fail in the IF statement, hence it would also fail in the added WHILE statement, and there would be no iterations of *d*. Thus

$$\boxed{\texttt{WHILE } b \texttt{ DO } d} =$$

$$\boxed{\texttt{IF } b \texttt{ THEN BEGIN } d\texttt{; WHILE } b \texttt{ DO } d \texttt{ END ELSE WHILE } b \texttt{ DO } d}.$$

Now, since WHILE *b* DO *d* is the last statement in both THEN and ELSE parts, it can be moved behind the IF statement, so that

$$\boxed{\texttt{WHILE } b \texttt{ DO } d} =$$

$$\boxed{\texttt{BEGIN IF } b \texttt{ THEN } d\texttt{; WHILE } b \texttt{ DO } d \texttt{ END}}.$$

The BEGIN statement meaning can be worked out directly, to get:

$$\boxed{\texttt{WHILE } b \texttt{ DO } d} = \boxed{\texttt{IF } b \texttt{ THEN } d} \circ \boxed{\texttt{WHILE } b \texttt{ DO } d}$$

The meaning of the WHILE statement itself has been separated out on the right side.

Now suppose a candidate function f is the correct meaning of the WHILE statement, that is,

$$f = \boxed{\texttt{WHILE } b \texttt{ DO } d}.$$

Then f must satisfy the equation for $\boxed{\texttt{WHILE } b \texttt{ DO } d}$ above:

$$f = \boxed{\texttt{IF } b \texttt{ THEN } d} \circ f.$$

Such an equation in which an unknown function occurs on both sides is called a *recurrence equation.*

A correct candidate function must satisfy the recurrence equation for its WHILE statement, but it does not follow that any function satisfying the equation is in fact the meaning of the WHILE statement. A simple case will show the pitfall. Consider a WHILE statement that never terminates:

```
WHILE 'A' = 'A' DO {nothing}
```

It is clear that this WHILE statement has a meaning that is empty—it contains no ordered pairs because the WHILE statement does not terminate on any input. But the recurrence reduces to

$$f = f$$

for this WHILE statement, since

$$\boxed{\texttt{IF 'A' = 'A' THEN \{nothing\}}}$$

is an identity function. Any function f satisfies $f = f$, yet all save the empty function are wrong for the WHILE statement.

A remedy for this problem is to add a condition ruling out extraneous solutions to the recurrence equation. The pitfall above can be avoided by not allowing candidate functions that are defined when the WHILE statement does not terminate. That is, we require that

domain(f) = domain(`WHILE b DO d`)

in addition to f satisfying the recurrence equation.

Another pitfall is contained in a WHILE statement that never iterates, for example:

```
WHILE 'A' <> 'A' DO {nothing}
```

In this case the WHILE statement has a meaning that is an identity function; it acts as a null statement. But, again, since

`IF 'A' <> 'A' THEN {nothing}`

is the identity function, the recurrence equation reduces to $f = f$, an equation any function satisfies. This WHILE statement has a meaning that is an identity function, but many non-identity functions satisfy both the recurrence equation and the domain constraint above. A condition that will permit only the identity function for this WHILE statement is the following:

For any execution state $s \in$ domain(`WHILE b DO d`),

if $\boxed{b}(s) = \mathbf{F}$ then $f(s) = s$.

That is, f must act as an identity on any state s where $\boxed{b}(s) = \mathbf{F}$. This condition can be expressed as the equality of two conditional assignment meanings:

$$(\text{NOT}(b) \rightarrow f) = (\text{NOT}(b) \rightarrow I)$$

where I is the identity function. Happily, just these two additional conditions ensure that a function satisfying the recurrence equation must be the meaning of the WHILE statement.

Thus the WHILE statement verification rule is:

To prove

$f =$ `WHILE b DO d`

it is necessary and sufficient to prove

1. domain(f) = domain(`WHILE b DO d`),
2. $(\text{NOT}(b) \rightarrow f) = (\text{NOT}(b) \rightarrow I)$,
3. $f =$ `IF b THEN d` $\circ f$.

8.2.2 Composition of Conditional Assignments

Conditional assignments are a powerful notation for expressing meanings of individual statements. They were invented to describe IF statements, but they are useful in analyzing WHILE statements, too. To check condition 3 of the WHILE statement verification rule requires calculating the composition of an IF statement meaning with the candidate function. Often the candidate function, and the function of the DO part in the WHILE statement, can be expressed as conditional assignments. Conditional trace tables provide a way to calculate the meaning of composed conditional assignments.

Let b and c be Boolean conditions, g and h be concurrent assignments, u and v be conditional assignments. The most general composition of two conditional assignments in reduced form is:

$$(b \rightarrow g \mid u) \circ (c \rightarrow h \mid v)$$

This function separates into four cases, corresponding to combinations of the guards:

$$\begin{aligned}
&(b \rightarrow g) \circ (c \rightarrow h) \\
&\cup\ (b \rightarrow g) \circ (\text{NOT}(c) \rightarrow v) \\
&\cup\ (\text{NOT}(b) \rightarrow u) \circ (c \rightarrow h) \\
&\cup\ (\text{NOT}(b) \rightarrow u) \circ (\text{NOT}(c) \rightarrow v)
\end{aligned}$$

If the conditional assignments are less complex, some of these cases may be missing. These terms can be evaluated with conditional trace tables, and the results recombined into a conditional assignment.

For example, suppose we wish to show that the meaning of

```
IF V1 < V2 THEN V1 := V2; V1 := V2
```

and

```
V1 := V2
```

are the same. The first meaning is

$$\boxed{\texttt{IF V1 < V2 THEN V1 := V2}} \circ \boxed{\texttt{V1 := V2}},$$

or in conditional-assignment form:

```
((V1 < V2 → V1 := V2) | ( )) ∘ (V1 := V2)
= (V1 < V2 → V1 := V2) ∘ (V1 := V2)
      ∪ (NOT(V1 < V2) → ) ∘ (V1 := V2)
```

Each term on the right side can be worked out in a conditional trace table:

Statement	Condition	V1
(V1 < V2 → V1 := V2)	V1 < V2	V2
V1 := V2		V2

Result: (V1 < V2 → V1 := V2).

Statement	Condition	V1
(NOT(V1 < V2) →)	V1 >= V2	
V1 := V2		V2

Result: (V1 >= V2 → V1 := V2).

Thus the result is

```
(V1 < V2 → V1 := V2) ∪ (V1 >= V2 → V1 := V2)
```

and since the guards are mutually exclusive, the whole can be given as a conditional assignment:

```
(V1 < V2 → V1 := V2) | (V1 >= V2 → V1 := V2)
```

That is, we have calculated

$$\boxed{\texttt{IF V1 < V2 THEN V1 := V2}} \circ \boxed{\texttt{V1 := V2}}$$

```
    = (V1 < V2 → V1 := V2) | (V1 >= V2 → V1 := V2).
```

The calculation was undertaken to determine if this function is the same as $\boxed{\texttt{V1 := V2}}$, conditional assignment (V1 := V2), which can be seen directly. However, in more complex cases the methods of Section 7.2.3 may be needed to compare the resulting conditional assignments. Using those methods in this example, to show that

```
(V1 := V2) = (V1 < V2 → V1 := V2) | (V1 >= V2 → V1 := V2)
```

falls under case 3 of section 7.2.3, and the result is immediate.

8.2.3 WHILE Statement Verification Examples

Consider the statement *w*:

```
WHILE V1 < V2
DO
  {V1,V2,V3 := V3,V4,V2}
```

with the candidate function:

```
f = (V1>=V2 → ) | (V3>=V4 → V1,V2,V3 := V3,V4,V2) |
    (V2>=V4 → V1,V2,V3 := V2,V4,V4) | (V1,V2,V3 := V4,V4,V4)
```

f is equivalent to:

```
(V1>=V2 →  ) |
(V1<V2 AND V3>=V4 → V1,V2,V3 := V3,V4,V2) |
(V1<V2 AND V3<V4 AND V2>=V4 → V1,V2,V3 := V2,V4,V4) |
(V1<V2 AND V3<V4 AND V2<V4 → V1,V2,V3 := V4,V4,V4)
```

Since f was constructed directly, unless a mistake was made, it should be correct. To illustrate the WHILE statement verification rule, we will verify that

$$f = \boxed{w}$$

by showing that:

1. domain(f) = domain($\boxed{w}$)
2. (V1>=V2 → f) = (V1>=V2 →)
3. $f = \boxed{\texttt{IF V1<V2 THEN \{V1,V2,V3 := V3,V4,V2\}}} \circ f$.

1. domain(f) = domain($\boxed{w}$).

f is defined for all values of V1, V2, V3, V4 since in its original form the last conditional assignment has no guard. By referring to the trace table of iterations in section 8.1.2, it can be seen that V1 and V2 both eventually have the original value of V4 and and w will terminate then, if not before.

2. (V1>=V2 → f) = (V1>=V2 →).

Rewrite (V1>=V2 → f) by distributing V1>=V2 in each guard of f:

```
(V1>=V2 AND V1>=V2 →  ) |
(V1>=V2 AND V1<V2 AND V3>=V4 → V1,V2,V3 := V3,V4,V2) |
(V1>=V2 AND V1<V2 AND V3<V4 AND V2>=V4 → V1,V2,V3 := V2,V4,V4) |
(V1>=V2 AND V1<V2 AND V3<V4 AND V2<V4 → V1,V2,V3 := V4,V4,V4)
```

Simplifying the guards results in

```
(V1>=V2 →  )
```

since in each of the last three guards V1>=V2 AND V1<V2 evaluates to **F**. Thus the left side is identical to the right side.

3. $f = \boxed{\texttt{IF V1<V2 THEN V1,V2,V3 := V3,V4,V2}} \circ f$.

The composition on the right side must be calculated and compared with f. Since the IF conditional assignment has two conditions and the f conditional assignment has four conditions, there are eight trace tables to consider.

Case 1

Part	Condition	V1	V2	V3
IF	V1<V2	V3	V4	V2
f	V3>=V4			

Condition: V1<V2 AND V3>=V4

Assignment: V1,V2,V3 := V3,V4,V2

Case 2

Part	Condition	V1	V2	V3
IF	V1<V2	V3	V4	V2
f	V3<V4 AND V2>=V4	V2	V4	V4

Condition: V1<V2 AND V3<V4 AND V2>=V4
Assignment: V1,V2,V3 := V2,V4,V4

Case 3

Part	Condition	V1	V2	V3
IF	V1<V2	V3	V4	V2
f	V3<V4 AND V2<V4 AND V4>=V4	V4	V4	V4

Condition: V1<V2 AND V3<V4 AND V2<V4 AND V4>=V4
Assignment: V1,V2,V3 := V4,V4,V4

Since V4>=V4 always takes the value **T**, it may be removed from the composite condition.

Case 4

Part	Condition	V1	V2	V3
IF	V1<V2	V3	V4	V2
f	V3<V4 AND V2<V4 AND V4<V4			

This is the empty function since V4<V4 always takes the value **F**.

Case 5

Part	Condition	V1	V2	V3
IF	V1>=V2			
f	V1>=V2			

Condition: V1>=V2
Assignment: V1,V2,V3 := V1,V2,V3

which is an identity function.

Case 6

Part	Condition	V1	V2	V3
IF	V1>=V2			
f	V1<V2 AND V3>=V4			

which is the empty function since V1>=V2 AND V1<V2 is **F**.

Cases 7 and 8 are also empty functions since they have the same contradiction. Assembling the nonempty functions yields

```
(V1<V2 AND V3>=V4 → V1,V2,V3 := V3,V4,V2) |
(V1<V2 AND V3<V4 AND V2>=V4 → V1,V2,V3 := V2,V4,V4) |
(V1<V2 AND V3<V4 AND V2<V4 → V1,V2,V3 := V4,V4,V4)
(V1>=V2 → V1,V2,V3 := V1,V2,V3)
```

which is identical to *f* as part 3 of the verification rule requires.

As a second example of using the verification rule, consider:

```
WHILE Ch <> ' '
DO
  BEGIN
    WRITE(Ch);
    READ(Ch)
  END
```

First, a candidate part function is needed. Informal methods are not always trustworthy for obtaining meanings, but here they are fine, because once a candidate is chosen, the rule can be used to check it. The loss in considering an incorrect candidate is only the effort of applying the rule

several times.

If Ch is not blank, then the WHILE statement will copy Ch and the next contiguous sequence of nonblank characters from INPUT to OUTPUT. If Ch is blank, then nothing happens. In order to express the meaning formally, consider the conditions under which the DO part of the statement will execute. First, the value of Ch will not be blank. Second, the input string must contain a blank in order for the loop to terminate. That is, INPUT_2 (the second list element of INPUT, its future string) should be of the form:

$(x \nabla c)\ \&\ y$,

where x is a possibly empty string of nonblank characters, c is a blank or line marker, and y is any string.

Third, the final value of Ch will be blank, INPUT will have been advanced over the blank following x, and OUTPUT will have the initial value of Ch, with the string x appended.

This information is conveniently expressed in a conditional assignment. Subscripted file identifiers indicate the corresponding components of their list values. Then the candidate meaning f is:

$$\begin{aligned}&((\mathrm{Ch} <> \text{ ' '})\ \mathrm{AND}\ (\mathrm{INPUT}_2 = (x \nabla c)\ \&\ y) \rightarrow \\ &\quad \mathrm{Ch},\ \mathrm{OUTPUT}_1,\ \mathrm{INPUT}_1,\ \mathrm{INPUT}_2 := \\ &\quad \square,\ (\mathrm{OUTPUT}_1 \nabla \mathrm{Ch})\ \&\ x,\ (\mathrm{INPUT}_1\ \&\ x) \nabla c,\ y)\ \ | \\ &(\mathrm{Ch} = \text{ ' '} \rightarrow\ )\end{aligned}$$

To show that this is in fact the meaning of the WHILE statement,

$f = \boxed{\texttt{WHILE Ch <> ' ' DO BEGIN WRITE(Ch); READ(Ch) END}}$

requires checking the three conditions of the verification rule:

1. $\mathrm{domain}(f) = \mathrm{domain}(\boxed{\texttt{WHILE ... END}})$,
2. $(\mathrm{Ch} = \text{ ' '} \rightarrow f) = (\mathrm{Ch} = \text{ ' '} \rightarrow I)$,
3. $f = \boxed{\texttt{IF Ch <> ' ' THEN BEGIN WRITE(Ch); READ(Ch) END}} \circ f$.

1. $\mathrm{domain}(f) = \mathrm{domain}(\boxed{\texttt{WHILE ... END}})$.

In the analysis that led to the candidate f, it was established that the WHILE statement terminates for nonblank values of Ch if the INPUT future string contains a blank or line marker; no repetitions occur if Ch has a blank value. The two conditional expressions in f define it for exactly these cases. Thus the domains of these functions are the same.

2. $(\mathrm{Ch} = \text{ ' '} \rightarrow f) = (\mathrm{Ch} = \text{ ' '} \rightarrow I)$.

The states on which the candidate f must be the identity are those in which Ch is blank, that is, those of the second guard in its definition. That is exactly how f is defined in that case.

3. $f = \boxed{\texttt{IF Ch <> ' ' THEN BEGIN WRITE(Ch); READ(Ch) END}} \circ f$.

First find the meaning of the IF statement, starting with its BEGIN statement. The WRITE statement computes the function expressed as a conditional assignment:

$$\boxed{\texttt{WRITE(Ch)}} = (\mathrm{OUTPUT}_1 := \mathrm{OUTPUT}_1 \nabla \mathrm{Ch}).$$

And for the meaning of the READ statement:

$$\begin{aligned}&\boxed{\texttt{READ(Ch)}} = (\mathrm{INPUT}_2 <> \dagger\dagger \rightarrow \\ &\quad \mathrm{Ch}, \mathrm{INPUT}_1, \mathrm{INPUT}_2 := \Theta\ \mathrm{INPUT}_2, \mathrm{INPUT}_1 \nabla \Theta(\mathrm{INPUT}_2), \Lambda(\mathrm{INPUT}_2)\ ).\end{aligned}$$

Combining these two statements, the conditional assignment of the BEGIN statement is:

$$\boxed{\texttt{BEGIN WRITE(Ch); READ(Ch) END}} =$$
$$(\texttt{INPUT}_2 \neq \dagger\dagger \rightarrow$$
$$\texttt{Ch}, \texttt{OUTPUT}_1, \texttt{INPUT}_1, \texttt{INPUT}_2 :=$$
$$\Theta\ \texttt{INPUT}_2, \texttt{OUTPUT}_1 \nabla \texttt{Ch}, \texttt{INPUT}_1 \nabla \Theta(\texttt{INPUT}_2), \Lambda(\texttt{INPUT}_2)).$$

The IF statement

```
IF Ch <> ' '
THEN
  BEGIN
    WRITE(Ch);
    READ(Ch)
  END
```

therefore has part function

$$g = \boxed{\texttt{IF Ch <> ' ' THEN BEGIN WRITE(Ch); READ(Ch) END}} =$$
$$(\texttt{Ch} <> \text{' '} \texttt{ AND } \texttt{INPUT}_2 <> \dagger\dagger \rightarrow$$
$$\texttt{Ch}, \texttt{OUTPUT}_1, \texttt{INPUT}_1, \texttt{INPUT}_2 :=$$
$$\Theta(\texttt{INPUT}_2), \texttt{OUTPUT}_1 \nabla \texttt{Ch}, \texttt{INPUT}_1 \nabla \Theta(\texttt{INPUT}_2), \Lambda(\texttt{INPUT}_2)) \ |$$
$$(\texttt{Ch} = \text{' '} \rightarrow\ ).$$

Next, g and f must be composed. There are four cases, each described by a conditional trace table. $\texttt{INPUT}_j$ is abbreviated I_j and $\texttt{OUTPUT}_j$ is abbreviated O_j in these tables.

Case 1

Part	Condition	Ch	$\texttt{OUTPUT}_1$	$\texttt{INPUT}_1$	$\texttt{INPUT}_2$
g	(Ch <> ' ') AND $(I_2 <> \dagger\dagger)$	$\Theta(I_2)$	$O_1 \nabla \texttt{Ch}$	$I_1 \nabla \Theta(I_2)$	$\Lambda(I_2)$
f	$(\Theta(I_2) <> \text{' '})$ AND $(\Lambda(I_2) = (x \nabla c)\ \&\ y)$	$\square$	$((O_1 \nabla \texttt{Ch}) \nabla \Theta(I_2)) \ \&\ x$	$(I_1 \nabla \Theta(I_2)) \ \&\ (x \nabla c)$	y

where x is a (possibly empty) string of nonblank characters, c is a blank or line marker, and y is any string.

Condition: $(\texttt{Ch} <> \text{' '}) \texttt{ AND } (\texttt{INPUT}_2 <> \dagger\dagger) \texttt{ AND } (\Theta(\texttt{INPUT}_2) <> \text{' '}$
$\texttt{AND } (\Lambda(\texttt{INPUT}_2) = (x \nabla c)\ \&\ y)$

Assignment: $\texttt{Ch}, \texttt{OUTPUT}_1, \texttt{INPUT}_1, \texttt{INPUT}_2 :=$
$\square,\ ((\texttt{OUTPUT}_1 \nabla \texttt{Ch}) \nabla \Theta(\texttt{INPUT}_2)) \ \&\ x,$
$(\texttt{INPUT}_1 \nabla \Theta(\texttt{INPUT}_2)) \ \&\ (x \nabla c),\ y$

At each domain point where $\Theta(\texttt{INPUT}_2)$ is defined, $\texttt{INPUT}_2 <> \dagger\dagger$ evaluates to $\mathbf{T}$. Therefore, the presence of $\Theta(\texttt{INPUT}_2) <> \text{' '}$ in the condition makes $\texttt{INPUT}_2 <> \dagger\dagger$ redundant. Furthermore,

$$\Theta(\texttt{INPUT}_2) <> \text{' '}$$
$$\Lambda(\texttt{INPUT}_2) = (x \nabla c)\ \&\ y$$

where x is a string of nonblank characters, imply that $\texttt{INPUT}_2$ can be written

$$(z \nabla c)\ \&\ y$$

where z is a nonempty string of nonblank characters. Thus the condition

$$\texttt{INPUT}_2 = (z \nabla c)\ \&\ y$$

can replace both of the other conditions on $\texttt{INPUT}_2$ in the guard, and occurrences of

$(\nabla\ \Theta(\mathtt{INPUT}_2))\ \&\ x$

in the assignment can be replaced by z.

Thus for Case 1, the part meaning is:

$$((\mathtt{Ch} <> \text{ ' '})\ \mathtt{AND}\ (\mathtt{INPUT}_2 = (z\ \nabla\ c)\ \&\ y) \rightarrow$$
$$\mathtt{Ch}, \mathtt{OUTPUT}_1, \mathtt{INPUT}_1, \mathtt{INPUT}_2 :=$$
$$\square, (\mathtt{OUTPUT}_1\ \nabla\ \mathtt{Ch})\ \&\ z, (\mathtt{INPUT}_1\ \&\ z)\ \nabla\ c, y).$$

Case 2

Part	Condition	Ch	$\mathtt{OUTPUT}_1$	$\mathtt{INPUT}_1$	$\mathtt{INPUT}_2$
g	(Ch <> ' ') AND $(\mathtt{I}_2$ <> ††)	$\Theta(\mathtt{I}_2)$	$\mathtt{O}_1\ \nabla$ Ch	$\mathtt{I}_1\ \nabla\ \Theta(\mathtt{I}_2)$	$\Lambda(\mathtt{I}_2)$
f	$\Theta(\mathtt{I}_2)$ = ' '				

Condition: $(\mathtt{Ch} <> \text{ ' '})\ \mathtt{AND}\ (\mathtt{INPUT}_2 <> \dagger\dagger)\ \mathtt{AND}\ (\Theta(\mathtt{INPUT}_2) = \text{ ' '})$

Assignment: $\mathtt{Ch}, \mathtt{OUTPUT}_1, \mathtt{INPUT}_1, \mathtt{INPUT}_2 :=$
$\Theta(\mathtt{INPUT}_2), \mathtt{OUTPUT}_1\ \nabla\ \mathtt{Ch}, \mathtt{INPUT}_1\ \nabla\ \Theta(\mathtt{I}_2), \Lambda(\mathtt{INPUT}_2)$

Once again, the presence of $\Theta(\mathtt{INPUT}_2)$ in the guard makes the condition $\mathtt{INPUT}_2$ <> †† redundant, $\mathtt{INPUT}_2$ has value $\square\ \&\ y$ for some string y, and the part meaning is:

$$((\mathtt{Ch} <> \text{ ' '})\ \mathtt{AND}\ (\Theta(\mathtt{INPUT}_2) = \text{ ' '} \rightarrow$$
$$\mathtt{Ch}, \mathtt{OUTPUT}_1, \mathtt{INPUT}_1, \mathtt{INPUT}_2 :=$$
$$\square, \mathtt{OUTPUT}_1\ \nabla\ \mathtt{Ch}, \mathtt{INPUT}_1\ \nabla\ \square, y)$$

Case 3

Part	Condition	Ch	$\mathtt{OUTPUT}_1$	$\mathtt{INPUT}_1$	$\mathtt{INPUT}_2$
g	(Ch = ' ') AND $(\mathtt{I}_2 = (x\ \nabla\ c)\ \&\ y)$				
f	Ch <> ' '				

The table has not been filled in, because the condition reduces to **F**; that is, this case cannot occur because Ch cannot be both blank and nonblank.

Case 4

Part	Condition	Ch	$\mathtt{OUTPUT}_1$	$\mathtt{INPUT}_1$	$\mathtt{INPUT}_2$
g	Ch = ' '				
f	Ch = ' '				

Condition: Ch = ' '

Assignment: ( )

This is identical to the second conditional assignment of f.

Collecting the part meanings for $g \circ f$:

$$((\mathtt{Ch} <> \text{ ' '})\ \mathtt{AND}\ (\mathtt{INPUT}_2 = (z\ \nabla\ c)\ \&\ y) \rightarrow$$
$$\mathtt{Ch}, \mathtt{OUTPUT}_1, \mathtt{INPUT}_1, \mathtt{INPUT}_2 :=$$
$$\square, (\mathtt{OUTPUT}_1\ \nabla\ \mathtt{Ch})\ \&\ z, (\mathtt{INPUT}_1\ \&\ z)\ \nabla\ c, y)\ |$$
$$((\mathtt{Ch} <> \text{ ' '})\ \mathtt{AND}\ (\Theta(\mathtt{INPUT}_2) = \text{ ' '} \rightarrow$$
$$\mathtt{Ch}, \mathtt{OUTPUT}_1, \mathtt{INPUT}_1, \mathtt{INPUT}_2 := \square, \mathtt{OUTPUT}_1\ \nabla\ \mathtt{Ch}, \mathtt{INPUT}_1\ \nabla\ \square, y)\ |$$
$$(\mathtt{Ch} = \text{ ' '} \rightarrow\ )$$

where z is a string of nonblanks, c is a blank or line marker, and y is a string. The second

conditional assignment can be seen to be a special case of the first (with $z = \dagger\dagger$ and $c = \square$), so it can be eliminated. The result is identical to f. That is, it has been shown that $f \circ g = f$ as condition 3 of the WHILE verification rule requires.

These sample verifications are more difficult than the calculations of Chapters 6 and 7, and have a fundamentally different nature. For iteration-free statements, the meaning of programs can be calculated directly from the definition. For WHILE statements things are not so nice. It is necessary to find the meaning by some informal means, then verify that it is correct. The situation is appropriate to the power of iteration. In practice, the need for a candidate function is not often a problem. A helpful comment often supplies it or intuitive understanding of the code can be used to work it out. Once the hypothesis is in hand, however, the program calculus establishes that it is or is not correct, with rigor equal to that used to derive the part functions of simpler constructions.

8.2.4 Exercises

8.2.1 Consider the recurrence equation:

$f = f \circ f$.

Find two different functions that satisfy this equation.

8.2.2 Which of the three conditions in the WHILE statement verification rule

a) is most difficult to establish?

b) requires the greatest knowledge about the WHILE statement, as opposed to the candidate function?

8.2.3 In each part, use the WHILE statement verification rule to show that the meaning of the given WHILE statement is the concurrent assignment given as a comment:

a)
```
{V1 := max(V1,V2)}
WHILE V1 < V2
DO
  V1 := V2
```

b)
```
{V1 := min(V1,V2)}
WHILE V2 < V1
DO
  V1 := V2
```

c)
```
{V2 := min(V1,V2)}
WHILE V2 < V1
DO
  V2 := V1
```

d)
```
{V2 := max(V1,V2)}
WHILE V2 <= V1
DO
  V2 := V1
```

8.2.4 For each WHILE statement given in Exercise 8.1.2, find a candidate function for the meaning, and use the WHILE statement verification rule to prove that your candidate is correct.

8.3 The Design of Iteration Statements

Preview

WHILE statements can be designed in a systematic way to satisfy an intended assignment. The design rules are derived from the WHILE statement verification rule.

A WHILE statement determines a repeated composition of its DO part dynamically, depending on its WHILE condition, just as a BEGIN statement determines a composition of its constituent statements statically. Thus, a WHILE statement is more complex to design than a BEGIN statement, but there are still important lessons to be learned from the BEGIN statement design rule.

8.3.1 WHILE Statement Design Rules

In design, an intended function f is given, and a WHILE statement W:

```
WHILE b
DO
  d
```

must be obtained. Since the meaning of W is to be f, this is exactly the situation of the WHILE statement verification rule:

$$f = \boxed{W} = \boxed{\texttt{WHILE } b \texttt{ DO } d}$$

if and only if

1. $\text{domain}(f) = \text{domain}(\boxed{W})$,
2. $(\text{NOT}(b) \rightarrow f) = (\text{NOT}(b) \rightarrow I)$,
3. $f = \boxed{\texttt{IF } b \texttt{ THEN } d} \circ f$.

A design rule can be derived from these three conditions.

1. $\text{domain}(f) = \text{domain}(\boxed{W})$.

The condition requires that b and d be designed so that W terminates for every state in domain(f).

2. $(\text{NOT}(b) \rightarrow f) = (\text{NOT}(b) \rightarrow I)$.

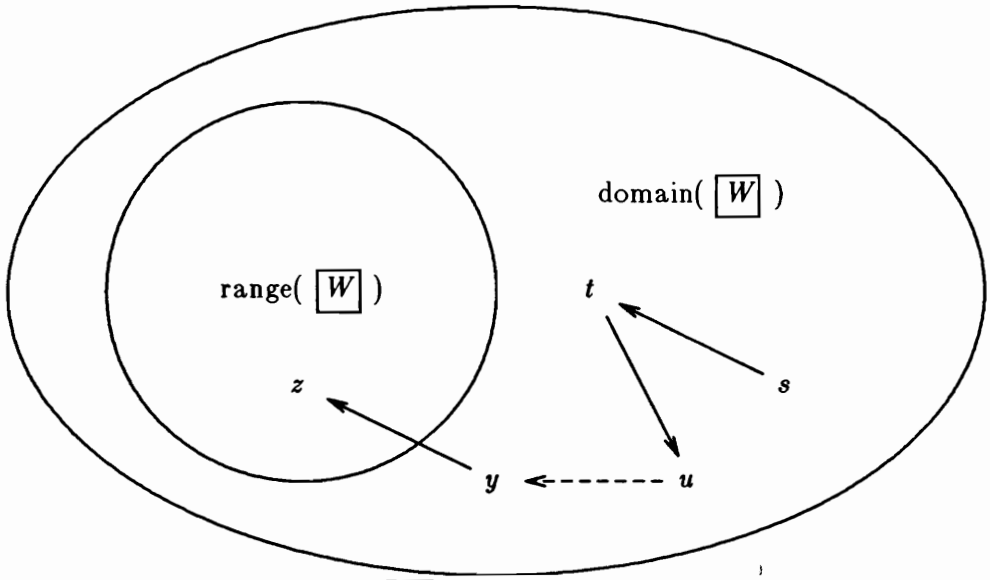
Since W terminates only when b evaluates to **F**, range($\boxed{W}$) is just those states for which b evaluates to **F**. Since any state for which b evaluates to **F** can also be an initial state,

$$\text{range}(\boxed{W}) \subseteq \text{domain}(\boxed{W}).$$

Therefore, f must have this property or no WHILE statement can have the desired meaning:

$$\text{range}(f) \subseteq \text{domain}(f).$$

The execution of WHILE statement W can be pictured as follows. Suppose the initial state s is in domain($\boxed{W}$) but not in range($\boxed{W}$). Then the state t after execution of the DO part may again be in domain($\boxed{W}$) - range($\boxed{W}$), and this may continue through many such states u, ..., y. If the execution terminates, it must do so because a state $z \in$ range($\boxed{W}$) is reached. On the other hand, if the initial state is in range($\boxed{W}$) the DO part is never executed at all, and the final state is unchanged.



Since b evaluates to **F** within range($\boxed{W}$), the WHILE statement W acts as a null statement in range($\boxed{W}$). Unless f has this same property—that is, if $s \in \text{range}(f)$, then $f(s) = s$, no correct WHILE statement exists.

These two requirements on the intended function f should therefore be checked before design begins, for if they do not hold, no WHILE statement can be designed:

range(f) $\subseteq$ domain(f), and

f behaves as the identity function in range(f).

The first step in the design itself is to determine the Boolean expression b. b must hold for any execution state in domain(f) - range(f); otherwise W would stop its iteration before reaching a proper state in range(f). b must not hold for any state in range(f); otherwise W would continue iteration instead of terminating when it reaches a state in range(f). Thus $\boxed{b}$ must divide into two cases such that its value is **T** on domain(f) - range(f) and **F** on range(f).

Once b has been derived from f, it remains to determine d. d must bring about the eventual termination of W from any state in which b holds, by transforming to a state in which b does not hold. In addition, condition 3 of the WHILE statement verification rule,

3. $f = \boxed{\texttt{IF } b \texttt{ THEN } d} \circ f$,

is the primary constraint on d. When b holds, if d preserves the expression values needed in f, then those values will be mapped the same by both sides of the equation. Thus, designing d to satisfy condition 3 is similar to designing the first statement of a BEGIN statement.

This discussion can be summarized in the following design rule for WHILE statements:

Given an intended meaning f:

1. (*Existence Rule*) Verify that:
   a. range(f) $\subseteq$ domain(f)
   b. if $s \in$ range(f) then $f(s) = s$.

otherwise no WHILE statement can be designed for f.

2. (*Boolean Condition Rule*) Determine b such that:
   a. $\boxed{b}$ evaluates to **T** in (domain(f) - range(f))
   b. $\boxed{b}$ evaluates to **F** in range(f).
3. (*DO Part Rule*) Determine d such that:

a. IF *b* THEN *d* preserves all values *f* requires to define the final state.

b. WHILE *b* DO *d* terminates for every state in domain(*f*).

For example, suppose the intended function is described by the concurrent assignment *f*:

```
V1,V2 := max(V1,V2),min(V1,V2)
```

For the existence rule, note that domain(*f*) is all values of V1 and V2, and range(*f*) is all values of V1 not preceding V2; hence

$$\text{range}(f) \subseteq \text{domain}(f)$$

as required. Also, for any state in the range(*f*), where V2 does not precede V1, the variables are properly ordered, and *f* does not alter them; that is, *f* is identity on range(*f*).

To satisfy the Boolean condition rule, *b* can be

```
V1 < V2
```

whose value is **F** on range(*f*), and **T** elsewhere. The design is therefore:

```
WHILE V1 < V2
DO
   d
```

where *d* is yet to be determined.

For the final step, the DO part rule, exchanging the values of V1 and V2 is a possibility. Should the exchange occur, termination is immediate. Exchanging preserves both

```
max(V1,V2)
min(V1,V2)
```

so the design is finished:

```
WHILE V1 < V2
DO
   {V1,V2 := V2,V1}
```

As another example, consider the problem of removing extra blanks. In Section 8.1.3, the required function *f* was defined as

```
((Ch = ' ') AND (INPUT = <x,y,R>) → Ch,INPUT := c,<x & u ∇ c, v, R>) |
((Ch <> ' ') OR (INPUT = <x,††,R>) →  )
```

where

$y = (u \nabla c)\ \&\ v$,

u is a (possibly empty) string of blanks,

c is a character other than blank or line marker, and

x, v are strings.

A WHILE statement will be designed using the WHILE statement design rules.

1. (Existence Rule)

   a. range(*f*) ⊆ domain(*f*)

Domain(*f*) is all values of Ch and INPUT, and range(*f*) is all nonblank values of Ch and values of INPUT where the future string is empty. Thus condition (a) is satisfied.

   b. if $s \in$ range(*f*), then $f(s) = s$

For any state in range(*f*), Ch is a nonblank character or the future string of INPUT is empty, cases in which *f* acts as the identity.

2. (Boolean Condition Rule)

   a. *b* evaluates to **T** in (domain(*f*) - range(*f*))

   b. *b* evaluates to **F** in range(*f*)

The Boolean condition

```
(Ch = ' ') AND (NOT EOF)
```

evaluates to **F** in range(f) and to **T** in domain(f) - range(f) and therefore satisfies both conditions.

3. (DO Part Rule)

   a. `IF b THEN d` preserves all values f requires to define the final state.

   b. `WHILE b DO d` terminates for every state in domain(f).

A statement d must be chosen that advances the WHILE statement closer to termination. This can be done by reading `INPUT` and ignoring blanks and line markers. `READ(Ch)` should work because it moves any state in which `Ch` is □ closer to a state in which either `Ch` is nonblank or `EOF` is **T**.

Thus the WHILE statement designed is:

```
WHILE (Ch = ' ') AND (NOT EOF)
DO
  READ(Ch)
```

which is exactly the WHILE statement from which function f was derived.

8.3.2 WHILE Statement Design from Concurrent Assignments

When the intended meaning is a concurrent assignment, a list of variables assigned a corresponding list of expressions, the design rules can be more simply stated.

The condition rule requires that b evaluate to **T** when at least one variable does not have the value of the corresponding expression. When all the required assignments have been achieved, b must evaluate to **F**. Since there are a fixed number of variables and expressions, this condition can be written directly using the `OR` operator.

The DO part rule requires that each of the expressions be preserved by each application of d.

For example, redoing the design for

```
V1,V2 := max(V1,V2),min(V1,V2)
```

the literal translation of the statement that some variable has not achieved its intended value is:

```
(V1 <> max(V1,V2)) OR (V2 <> min(V1,V2))
```

or:

```
(V1 < V2) OR (V2 > V1)
```

which simplifies to

```
V1 < V2
```

as before.

8.3.3 Exercises

8.3.1 Does the existence rule for WHILE statement design allow the following intended function to be implemented as a WHILE statement?

```
(V1 ∈ {0,1,2,3,4,5,6,7,8,9} →
    V1 := next character after V1 in the string †01234567890†) |
(  )
```

8.3.2 Design WHILE statements for the following intended functions, or explain why they cannot be implemented as WHILE statements.

a) `V1,V2 := max(V1,V2),max(V1,V2)`

b) `V1,V2 := max(V1,V2,V3),min(V1,V2,V3)`

c) `V1,V2 := V2,V1`

d) `V1,V2,V3 := max(V1,V2,V3),min(V1,V2,V3),min(V1,V2,V3)`

8.3.3 Given any statements S, the WHILE statement

```
WHILE New = 'Y'
DO
  BEGIN
    S;
    New := 'N'
  END
```

will execute those statements, where New is chosen to be a variable that does not occur in *S*. Reconcile this fact with the assertion that some intended functions cannot be realized with a WHILE statement.

8.4 The Theory of Iteration Statements

Preview

The proof of the WHILE statement verification rule is by contradiction, so an example of a contradiction proof by Euclid is given first. Finally, it is shown that there is no general way to decide whether an arbitrary WHILE statement terminates for any initial condition.

8.4.1 A Proof By Contradiction

In mathematics, some problems are *impossible* to solve. For example, Euclid's proof that there is *no* fraction whose square is the natural number 2 is given below. Thus the problem of finding a fraction whose square is 2 has no solution. It is not that the problem is difficult to solve, or that no one knows if it can be solved; there is a proof that it *cannot* be solved. The proof that a problem has no solution is often by contradiction.

To show that no fraction can be the square root of 2, Euclid supposed to the contrary that the fraction with numerator p and denominator q had the square 2, that is, $(p/q)(p/q) = 2$. Let p/q be in lowest terms, where p and q have no common factors. Multiplying twice by q,

$$p \times p = 2 \times q \times q.$$

Since the right side is divisible by 2, the left side must also be (otherwise one would be odd and the other even), and that means that p itself is divisible by 2. It can therefore be written as $2\times(p\div 2)$, and the equation rewritten as

$$4 \times (p\div 2) \times (p\div 2) = 2 \times q \times q.$$

Dividing both sides by 2,

$$2 \times (p\div 2) \times (p\div 2) = q \times q.$$

This equation is in the same form as the original, but with q playing the role of p, and $(p\div 2)$ the role of q. Hence by the same argument, q must be divisible by 2. But recall that p was also divisible by 2, and at the same time the fraction was initially in lowest terms. This is a contradiction, because removing the common factor would reduce the fraction. Hence the assumption that there is a fractional square root of 2 must be in error, and that is what Euclid wanted to prove.

The form of the proof is simple: first assume the opposite of what is to be proved (in this case that there is the fraction p/q); manipulate this fact and others that are available (here by multiplying out and reasoning about what divisors terms must have) until a contradiction is obtained. Then the original assumption is at fault, and the proof is complete.

8.4.2 The WHILE Statement Verification Rule

The WHILE statement verification rule is an if-and-only-if statement, asserting that when when two functions are equal, three conditions hold; and, if the three conditions hold, the functions are equal:

$f = \boxed{W}$

where W is

WHILE b DO d

if and only if

1. domain(f) = domain($\boxed{W}$),
2. (NOT(b) $\rightarrow f$) = (NOT(b) $\rightarrow I$),
3. $f = \boxed{\text{IF } b \text{ THEN } d} \circ f$.

It is straightforward to show that the three conditions follow from the equality of the functions; the difficult part of the proof (which we will do by contradiction) is to show that when the three conditions hold, the functions must be equal. (And of course it is this difficult direction that is utilized when the rule is used for a verification.)

We begin with the easy part of the proof. Suppose that $f = \boxed{W}$. Then conditions 1, 2, and 3 can be demonstrated.

1. domain(f) = domain($\boxed{W}$).

When two functions are the same, so are their domains.

2. (NOT(b) $\rightarrow f$) = (NOT(b) $\rightarrow I$).

This equation is true with $\boxed{W}$ replacing f, by definition of the WHILE statement meaning. But since $f = \boxed{W}$, it holds for f as well.

3. $f = \boxed{\text{IF } b \text{ THEN } d} \circ f$.

In the definition of $\boxed{W}$ add an ELSE part that has no effect, then separate out the statement W and replace it with f:

$$f = \boxed{W} = \boxed{\text{IF } b \text{ THEN BEGIN } d;\ W \text{ END}}$$
$$= \boxed{\text{IF } b \text{ THEN BEGIN } d;\ W \text{ END ELSE } W}$$
$$= \boxed{\text{IF } b \text{ THEN } d} \circ \boxed{W}$$
$$= \boxed{\text{IF } b \text{ THEN } d} \circ f.$$

Thus if $f = \boxed{W}$ then conditions 1, 2, and 3 hold.

Conversely, suppose that the three conditions hold. The proof that $f = \boxed{W}$ is by contradiction, so assume the opposite:

$$f \neq \boxed{W}.$$

From this assumption the contradiction that W executes endlessly for an element of domain($\boxed{W}$) will be derived. Because f and $\boxed{W}$ differ, they must do so on at least one point of their common domain, by condition 1. Let this point be s.

What is the value of $\boxed{b}(s)$? Since s is in domain($\boxed{W}$), $\boxed{b}$ must be defined there, and it cannot be **F**. If $\boxed{b}(s)$ were **F**, by condition 2 f would be identity, that is

$$f(s) = s.$$

But by definition:

$$\boxed{W}(s) = \boxed{\text{IF } b \text{ THEN BEGIN } d;\ W \text{ END}}(s) = s.$$

Thus if $\boxed{b}(s)$ were **F**, f and $\boxed{W}$ would agree at s, and they do not, so $\boxed{b}(s)$ must be **T**. This means that

$$\boxed{W}(s) = \boxed{\text{IF } b \text{ THEN BEGIN } d;\ W \text{ END}}(s)$$
$$= \boxed{\text{BEGIN } d;\ W \text{ END}}(s)$$

$= \boxed{W}(\boxed{d}(s))$.

And it also means that, by condition 3,

$f(s) = (\boxed{\texttt{IF } b \texttt{ THEN } d} \circ f)(s)$

$= (\boxed{d} \circ f)(s)$

$= f(\boxed{d}(s))$.

That is, as a consequence of $f(s) \neq \boxed{W}(s)$, we have

$f(\boxed{d}(s)) \neq \boxed{W}(\boxed{d}(s))$.

The same argument may then be repeated, and repeated again without end, with the result that

$\boxed{b}(\boxed{d}(s)) = \mathbf{T}$

$\boxed{b}(\boxed{d}^2(s)) = \mathbf{T}$

$\boxed{b}(\boxed{d}^3(s)) = \mathbf{T}$

...

That is, W never terminates beginning with state s. But this contradicts the choice of s as in domain($\boxed{W}$).

Thus we have shown by contradiction that there can be no point s on which f and $\boxed{W}$ disagree, and that proves the difficult part of the verification rule.

8.4.3 The Undecidability of WHILE Statement Termination

WHILE statements allow a single program to process data whose extent is unlimited. For example, by iterating a READ statement, all the characters of a file can be processed. But this potential for unlimited repetition carries a corresponding drawback: the iteration may never terminate. WHILE statements can lead to endless execution. It would be very valuable to be able to analyze programs for termination. Ideally, the analysis would be mechanical, itself done by a program. If it were possible to decide in some general way if a program's WHILE statements could fail to terminate, a great deal of wasted computer time would be saved.

Unfortunately, automatic analysis for termination is *not* possible, as we will now prove. The proof is by contradiction, and it uses an idea fundamental to computer science—the idea of *self reference*. The essence of the proof is to imagine that an automatic termination-analyzing program *did* exist, and to think about asking that program to analyze itself. It can be arranged that a contradiction arises: the program can be made to execute endlessly as a consequence of deciding that it terminates.

The idea of programs that analyze other programs was once a radical one, but today it is commonplace. The whole of programming is built on such a base, because programs written in languages like Pascal are translated into machine terms by a compiler, a program-analyzing program. Compilers are complex programs, but other analysis programs are easy to imagine and understand. For example, a "pretty printing" program that creates a standardized listing for a CF Pascal program is not difficult to write. The pretty-printer looks for Pascal key words, and prints lines so that the key words begin them with the proper indentation.

To show that there is no way in general to learn if a program will terminate or execute endlessly for a given input, make the assumption that such an analysis *is* possible. Suppose that there were a CF Pascal procedure to do it, which might look like this:

```
PROCEDURE Terminate;
  {Analyze the contents of F1 as a CF Pascal program
   and F2 as its input, setting the variable Result
   to 'T' or 'N' depending on whether or not
   the program in F1 will terminate in execution with
   the data in F2 as input.  Restore F1 and F2 to their
   original condition.}
  BEGIN
    ...
  END
```

It might be expected that the body of `Terminate` would be difficult to write, and would use many clever ideas. Perhaps its text would be very long, and would require a great deal of execution time. Nonetheless, if it is possible to do the task at all, the comment on `Terminate` would be realized.

But `Terminate` *cannot* be written, no matter how clever we are. For if it could, it would be possible to turn it on itself using the following design:

```
PROGRAM OhYeah(INPUT,OUTPUT);
  VAR
    F1, F2: TEXT;
    Result: CHAR;
  PROCEDURE Terminate;
    {Analyze the contents of F1 as a CF Pascal program
     and F2 as its input, setting the variable Result
     to 'T' or 'N' depending on whether or not
     the program in F1 will terminate in execution with
     the data in F2 as input.  Restore F1 and F2 to their
     original condition.}
    BEGIN
      ...
    END;
  BEGIN {OhYeah}
    {Copy INPUT to F1};
    {Copy F1 to F2 and reset both files};
    Result := 'T';
    WHILE Result = 'T'
    DO
      Terminate;
    WRITELN(Result)
  END.
```

This program uses the procedure `Terminate` on its input, but in a somewhat peculiar way. It can be seen that the only result it can write is **N**, since if `Terminate` returns **T**, the WHILE statement will keep going. Thus `OhYeah` itself will not terminate if it is given input for which `Terminate` returns **T**. The other peculiar feature of `OhYeah` is that it treats the input as both program and data; that is, it sets `Terminate` to analyzing a program working on its own text as input. Perhaps `OhYeah` isn't a very sensible program. Its action is to print **N** if the input, treated as a CF Pascal program working on its own text, would fail to terminate; otherwise, `OhYeah` executes endlessly. All this supposes that `Terminate` can be written, of course. But `OhYeah` wasn't written to be sensible, but to lead to a contradiction, in order to show that `Terminate` *cannot* be written. The contradiction appears if we ask the question:

What will `OhYeah` do if it is given as input a copy of its own text?

Assuming that `Terminate` can be filled in, `OhYeah` is a CF Pascal program, so this question must have an answer. Perhaps `OhYeah` fails to terminate for that case, or perhaps it prints **N**, the two possibilities. But in fact neither of these can occur. Suppose `OhYeah` prints **N**. Then the input, with its own text as data, fails to terminate. That input was `OhYeah`, and it *did* terminate, a

contradiction. On the other hand, if `OhYeah` fails to terminate with this input, it means that `Terminate` is returning the value **T** repeatedly, that is, `OhYeah` has been analyzed to terminate with input `OhYeah`. But that is also a contradiction.

The only assumption made in creating `OhYeah` is that `Terminate` could be written to analyze program termination in general. Hence that assumption is false, and the general problem of termination is *unsolvable*. In particular cases, this result has no force. No program can be designed to correctly analyze *all* programs for termination. But an analysis of any individual program can be attempted, and enough hard work may lead to a proof of termination. The WHILE statement verification theorem requires (in the domain condition 1) that programs be subjected to this analysis, and that it succeed. In practice, the program, the intended function, the programmer can be changed until the proof is realized.

8.4.4 Exercises

8.4.1 If a CF Pascal procedure were written to pretty-print CF Pascal programs and procedures, could it be used to pretty-print itself? If you decide it can, explain why an `OhYeah` program cannot be written to produce a contradiction. If you decide no, outline your proof.

8.4.2 Make up a paragraph to precede the sentence "This sentence is false." which creates a dilemma of undecidability.

8.4.3 Since CF Pascal syntax has been given with recursion, e.g.,

<identifier> ::= <letter> ¦ <identifier> <letter> ¦ <identifier> <digit>

is the problem of deciding if a given string can be obtained from a given syntax rule unsolvable? Explain your conclusion.

8.4.4 In the discussion about `Terminate` and `OhYeah`, could the program data be eliminated from the discussion by making it a constant? That is, could `Terminate` treat `F1` as a program, and decide termination for the fixed input string (say) †**TEST**†? If so, what would have been proved?

8.4.5 Why is it necessary to restore `F1` and `F2` in the supposed `Terminate` procedure?

8.4.6 It is iteration (and recursion) that causes the problem of termination to be undecidable.

a) Explain why CF Pascal programs without iteration (and without recursion) can be analyzed to learn if they terminate.

b) Explain how iteration enters the proof in Section 8.4.3. That is, why can't the same proof be used to show that termination for CF Pascal programs without iteration is unsolvable?

8.5 Chapter Summary

The meaning of a WHILE statement can be defined recursively to follow the meaning expressed in execution tables, in which the entire statement is repeated after each successful iteration. The execution will terminate only if the iteration count is finite, leading to an alternate expression of WHILE statement meaning.

Formal analysis of WHILE statements is more difficult than the other parts of the program calculus. Instead of calculating meaning directly it is necessary to obtain it independently, then check its correctness. The WHILE statement verification rule provides a way to perform this check.

The design of WHILE statements is also difficult, but rules that combine insights about the role of the Boolean condition with those for designing BEGIN statements were presented.

In general it is impossible to systematically predict the termination of programs, and this undecidability is traceable to the termination of WHILE statements.

PART III: DEVELOPING LARGER PROGRAMS

Most of the fundamental ideas of good programming have been presented, and the program calculus given for most of CF Pascal. However, just as the novice typist may find some newly learned typing skills awkward and difficult to use, so you may find it easier to attack a programming problem directly, without using the program calculus. Its ease and naturalness of use will come only with practice. For this reason you must design, develop, and analyze programs using the program calculus even though it may be clumsy at first. Programs should be developed from their functions and analyzed by abstracting out their functions. As new ideas about problem solving and programming are presented, you can take the opportunity to practice and refine the basic skills learned in the first two parts of this text. Just as with touch typing, in time the skill will become second nature and you will be able to solve problems that would be impossible by direct, hunt-and-peck programming.

Procedures and Libraries

As problems get larger, the programs to solve them must be made of independent parts. This is because the whole can be understood only by analyzing one part at a time, and parts must be used more than once—both within a single program, and in different programs. Once a standard program part has been developed to implement a function, it can be used wherever that function is needed. Once it has been proved that the program part has been designed correctly, the analysis need not be repeated each time it is used. The gain in intellectual control that comes from breaking a problem into independent parts is so substantial that large tasks cannot be accomplished without it.

Procedures are the CF Pascal mechanism for dividing a program into independent parts. The simple procedures introduced in Section 2.4 will be extended to more closely represent functions, so that they can be tailored to a wide variety of applications. By grouping procedures in "libraries" dealing with a single subject, the programmer can more easily find the standard parts needed for each problem. Since procedures in the libraries can be used over and over again, it is important to correctly derive each procedure from the specified function, and to document the function well so that it can be understood at a later date, possibly by another programmer. Libraries of procedures aid program development because they represent higher-level functions than those supplied by the programming language. Using these more abstract functions, the programmer can think more about the problem and less about details of the programming language. The fewer ideas a person has to juggle the more likely a correct solution to a problem will be found.

As examples of useful libraries, we will develop procedures for string processing, for operating with natural numbers, and for text manipulation.

Recursion

The most powerful and general problem-solving method is called "divide and conquer." The idea is to solve a problem by reducing it to one or more simpler problems, then in turn reducing each of those, and so on, until ultimately each problem is trivial to solve, and the solutions can be combined. Recursion is the most elegant form of divide-and-conquer programming. A procedure is written to solve part of the problem, and its central statement is an invocation of itself, for a smaller version of the same problem. When the procedure is invoked to handle a large problem, it thus repeatedly calls itself until the problem can be trivially solved. Recursion seems almost magic, because the solution to all problems and subproblems is written just once.

Recursive CF Pascal procedures will be an important programming technique.

Specification and Correctness

A program derived correctly from a problem statement should solve that problem, and we have emphasized that at each step of the programming process the program calculus should be used to verify that the correspondence between problem and solution is being maintained. However, the programming process can never be perfect, and many programs are designed for one purpose, then

modified for another. When program and problem statement become separated, the question arises of whether or not the two fit together. A problem statement divorced from a program is called a *specification*; it may be a relation giving many acceptable solutions rather than a function indicating just one. When a program P agrees with a specification S, we say that P is *correct* with respect to S.

In these terms, program development has the goal of starting with a specification and deriving a program that is correct with respect to it. If program P and specification S are given separately, the proof that P is correct with respect to S must be constructed, and its first step is to compute $\boxed{P}$ for comparison with S.

CHAPTER 9

CF PASCAL PROCEDURES

Chapter Preview

CF Pascal procedures are much more flexible and powerful than the simple ones introduced in Chapter 2. A procedure may have parameters that tailor it to many applications, making a single text serve many times. A procedure may declare internal variables, whose values are available only within the procedure, and thus protected from any external access. To illustrate the use of procedures, a small library of operations on natural numbers will be developed.

Procedures are the most important part of CF Pascal (and many other programming languages) because they allow a program to be broken into small sections for understanding. Each section is largely self-contained and can therefore be understood in isolation. When a procedure is used, it is unnecessary for the person using it to read the full declaration; instead, it is usually enough to go by the mnemonic meaning of the procedure name, or to read a summarizing comment. In a way, the procedure is the idea behind part functions: a portion of a program's effect can be captured and proved in a part function, then this meaning can be used in place of the text for understanding. The advantage of procedures over other CF Pascal constructions is that they may include arbitrary text, and thus have a greater power to abstract and summarize. Furthermore, the mechanism of parameters allows one procedure to generalize for many tasks, with a further saving in intellectual effort.

9.1 Procedures with Parameters

Preview

Procedures can be declared with formal parameters, so that the text stored in the execution state may be adapted to many different procedure statements, in which those parameters are replaced by different variables.

New ideas: parameters, all-or-nothing rule.

9.1.1 Simple Procedures and Parameters

The procedures described in Chapter 2 and given formal meaning in Chapter 6 are a mechanism to declare a CF Pascal BEGIN statement, then invoke it with a procedure statement using only the procedure identifier. The declaration adds the identifier to the execution state, with the value of the body text; the procedure statement has the same meaning as this text value.

For example, the declaration

```
PROCEDURE SwitchOneTwo;
  BEGIN
    Hold := OneV;
    OneV := TwoV;
    TwoV := Hold
  END
```

might be written to exchange the values of `OneV` and `TwoV`, destroying the value of `Hold`. This declaration adds the pair

<SwitchOneTwo, Hold := OneV; OneV := TwoV; TwoV := Hold>

to the state, and a following procedure statement

SwitchOneTwo

has the meaning

$$\boxed{\texttt{SwitchOneTwo}} = \langle s, \boxed{s(\texttt{SwitchOneTwo})}(s)\rangle$$

$$= \langle s, \boxed{\texttt{Hold := OneV; OneV := TwoV; TwoV := Hold}}(s)\rangle$$

$$= \texttt{(Hold, OneV, TwoV := OneV, TwoV, OneV)}$$

expressed as a conditional assignment. This part function is the meaning of any procedure statement using SwitchOneTwo, so once it has been calculated, it can be used whenever the procedure is called.

The drawback in using simple procedures is that all their data storage must come from the surrounding context. In the example, SwitchOneTwo interchanges just two particular variables, using a particular temporary variable; to change to any other variables requires another procedure. Parameters are the mechanism that eliminates this restriction.

Parameter identifiers are placed in the procedure header, and throughout its text they are "dummies," that is, they are merely placeholders for the variables that will actually be used in a subsequent procedure statement. For example, the header

```
PROCEDURE SwitchThese2(VAR V1, V2, Temp: CHAR)
```

has three ***formal parameters*** with character values. The body of the procedure may use these, say as follows:

```
PROCEDURE SwitchThese2(VAR V1, V2, Temp: CHAR);
  BEGIN
    Temp := V1;
    V1 := V2;
    V2 := Temp
  END
```

Procedure statements may now use the procedure identifier, followed by ***actual parameters*** that are any variables (of the same type as the formal parameters) that are in existence at the position of the procedure statement. For example, if variables Hold, OneV, and TwoV have been declared in a program, the procedure statement

```
SwitchThese2(OneV, TwoV, Hold)
```

will exchange values just as did the original simple procedure. But in addition,

```
SwitchThese2(OneV, AnyV, Hold)
```

will exchange the values of OneV and AnyV, without any change in the declaration of SwitchThese2. It is as if the text in the declaration has the actual variables substituted for the formal ones. In the last example,

$$\boxed{\texttt{SwitchThese2(OneV, AnyV, Hold)}}$$

$$= \boxed{\texttt{Hold := OneV; OneV := AnyV; AnyV := Hold}}$$

$$= \texttt{(Hold, OneV, AnyV := OneV, AnyV, OneV)}.$$

Procedures can have parameters of more than one type. For example,

```
PROCEDURE Copy(VAR F1, F2: TEXT; VAR Ch: CHAR);
```

could be used as a procedure heading whose BEGIN statement copies file F1 into file F2 preserving the line structure. (Note that VAR is required with each type of parameter.)

```
PROCEDURE Copy(VAR F1, F2: TEXT; VAR Ch: CHAR);
  BEGIN  {Copy}
    WHILE NOT EOF(F1)
    DO
      BEGIN
        WHILE NOT EOLN(F1)
        DO
          BEGIN
            READ(F1, Ch);
            WRITE(F2, Ch)
          END;
        READLN(F1);
        WRITELN(F2)
      END
  END  {Copy}
```

This procedure could then be used with any existing files, which substitute for the dummies `F1` and `F2`, and using any existing variable in place of `Ch`. For example,

```
Copy(INPUT, OUTPUT, Temp)
```

would echo the input to the output.

9.1.2 Procedure Parameter Syntax and Context Rules

To incorporate procedures with parameters into the syntax rules of CF Pascal requires changing Syntax Rules 27 and 28 and adding three rules:

SR27. <procedure> ::= <procedure heading> ; <BEGIN statement>

SR28. <procedure statement> ::= <identifier> ¦
<identifier> (<actual parameter list>)

SR30. <actual parameter list> ::= <identifier list>

SR31. <procedure heading> ::= PROCEDURE <identifier> ¦
PROCEDURE <identifier> (<formal parameter list>)

SR32. <formal parameter list> ::= VAR <identifier list> : <type> ¦
<formal parameter list> ; VAR <identifier list> : <type>

The context rules must also be modified:

CR4. The procedure identifier used in a <procedure statement> must appear in a <procedure heading> of a <procedure> declaration. The <actual parameter list> of a <procedure statement> must agree in length with the <formal parameter list> of the associated <procedure heading>. The identifiers in the <formal parameter list> and the <actual parameter list> must agree in type.

CR6. Any identifier declared in a procedure's <formal parameter list> may only be referenced in the <BEGIN statement> of the procedure. Such identifiers are said to be *local* to the procedure.

9.1.3 Splitting Files

In Section 4.1.5 a program was designed to transfer the input to the output, but with the even-numbered characters preceding the odd-numbered ones. A procedure with parameters can simplify this design. In addition the line structure of the input is handled explicitly, instead of using an end marker #. The top-level design was:

```
PROGRAM Split(INPUT,OUTPUT);
  {Copy INPUT to OUTPUT, odd-numbered characters first,
   even-numbered characters last}
  VAR
    Odds, Evens: TEXT;
  BEGIN  {Split}
    {Split INPUT between Odds and Evens};
    {Copy Odds to OUTPUT};
    {Copy Evens to OUTPUT};
    WRITELN
  END.  {Split}
```

The two tasks

```
{Copy Odds to OUTPUT}
{Copy Evens to OUTPUT}
```

can be accomplished by a common procedure CopyOut, if there is a parameter for which Odds and Evens can be substituted:

```
PROGRAM Split(INPUT,OUTPUT);
  {Copy INPUT to OUTPUT, odd-numbered characters first,
   even-numbered characters last}
  VAR
    Ch: CHAR;
    Odds, Evens: TEXT;
  PROCEDURE CopyOut(VAR F1: TEXT; VAR Ch1: CHAR);
    {Copy F1 to OUTPUT, ignoring line markers};
  BEGIN {Split}
    {Split INPUT between Odds and Evens};
    CopyOut(Odds, Ch);
    CopyOut(Evens, Ch);
    WRITELN
  END.  {Split}
```

Here is the definition of CopyOut:

```
PROCEDURE CopyOut(VAR F1: TEXT; VAR Ch1: CHAR);
  {Copy F1 to OUTPUT, ignoring line markers}
  BEGIN {CopyOut}
    RESET(F1);
    WHILE NOT EOF(F1)
    DO
      BEGIN
        WHILE NOT EOLN(F1)
        DO
          BEGIN
            READ(F1, Ch1);
            WRITE(Ch1)
          END;
        READLN(F1)
      END
  END   {CopyOut}
```

The BEGIN statement for CopyOut only uses the parameters F1 and Ch1 declared in the procedure heading. In contrast, a procedure without parameters makes use of variables declared in the program containing the procedure. A procedure with parameters could employ a mixture—some parameter variables and some from the surrounding program. However, procedures that use only their own parameters are more easily understood in isolation and are easier to reuse. The statement

that it is better to use only parameters is called the *all-or-nothing* rule.

9.1.4 The Calculus of Parameters

The text of a procedure's BEGIN statement is stored in the execution state as the value of the procedure identifier. This text may contain references to the formal parameters. Suppose for a moment that variables with the same identifiers as the formal parameters are used as the actual parameters in a procedure statement. Then the definition for simple procedures would give the right result. For example,

```
PROGRAM Sample(INPUT,OUTPUT);
  VAR
    Vin: CHAR;
  PROCEDURE Bump(VAR Vin: CHAR);
    BEGIN {Bump}
      IF Vin = 'A'
      THEN
        Vin := 'B'
    END;  {Bump}
  BEGIN {Sample}
    Vin := 'Q';
    Bump(Vin)
  END.  {Sample}
```

has

$$\boxed{\texttt{PROCEDURE Bump ... END}}(s)$$

$$= s \cup \{<\dagger\textbf{Bump}\dagger, \dagger\textbf{BEGIN IF Vin = 'A' THEN Vin := 'B' END}\dagger>\}$$

so the simple definition would give

$$\boxed{\texttt{Bump(Vin)3}}(s) = \boxed{s\,\texttt{(Bump)}}(s)$$

$$= \boxed{\texttt{BEGIN IF Vin = 'A' THEN Vin := 'B' END}}(s)$$

which is the correct meaning.

When the part function is calculated in isolation, from the declaration alone, parameter identifiers appear within the result. In the procedure Bump above,

$$\boxed{\texttt{IF Vin = 'A' THEN Vin := 'B'}}$$

$$= (\texttt{Vin = 'A'} \rightarrow \texttt{Vin := 'B'}) \mid (\ )$$

expressed as a conditional assignment. The identifier Vin is a "parameter" of this function, in the mathematical sense.

Formal and actual parameters are not always the same, but the same definitions would work if when the value of the procedure text is used, it had the actual parameters inserted for the formal ones. That is, if in the example the procedure statement were

```
Bump(DiffV)
```

(where the actual parameter DiffV is suitably declared CHAR), then the meaning would be obtained by inserting DiffV everywhere Vin appears in the text value of Bump. Thus

$$\boxed{\texttt{Bump(DiffV)}} = \boxed{\texttt{BEGIN IF DiffV = 'A' THEN DiffV := 'B' END}}.$$

In the general case, if P is a procedure identifier, and its declaration has formal parameter X, then with actual parameter Y, define

$$\boxed{P\ (\ Y\ )}(s) = \boxed{s\,(P)_{X \leftarrow Y}}(s)$$

where attaching the subscript $X \leftarrow Y$ means that within $s(P)$ each occurrence of X is replaced by Y. The generalization to more than one parameter is obvious: each actual-parameter identifier replaces the corresponding formal-parameter identifier.

The meaning of the procedure statement for Bump above can also be obtained from the "parameterized" conditional assignment by substituting the actual parameter. That is,

$$\boxed{\texttt{Bump(DiffV)}} = (\texttt{Vin = 'A'} \rightarrow \texttt{Vin := 'B'})_{\texttt{Vin} \leftarrow \texttt{DiffV}} \mid (\ ) = (\texttt{DiffV = 'A'} \rightarrow \texttt{DiffV := 'B'}) \mid (\ ).$$

But there is a subtle flaw in making this substitution in the general case. It can happen that a confusion in identifiers gives the wrong meaning. For example, the program

```
PROGRAM Alias(INPUT,OUTPUT);
  VAR
    One, Two: CHAR;
  PROCEDURE SwitchThese2(VAR V1, V2, Temp: CHAR);
    {V1,V2,Temp := V2,V1,V1}
    BEGIN {SwitchThese2}
      Temp := V1;
      V1 := V2;
      V2 := Temp
    END;  {SwitchThese2}
  BEGIN {Alias}
    SwitchThese2(One, Two, Two)
  END.  {Alias}
```

has (from the declaration)

$$\boxed{\texttt{SwitchThese2(One, Two, Two)}} = \boxed{\texttt{Two := One; One := Two; Two := Two}} = (\texttt{Two := One})$$

but substituting in the conditional assignment worked out from the declaration gives the wrong result:

```
(Two,Two,One := One,Two,Two) = (Two,One := One,Two).
```

This difficulty with parameter names is called "aliasing," because one actual identifier has two formal names; in the example, Two has the distinct names V2 and Temp.

When aliasing occurs, and the parametrized definition gives the wrong result, it is usually because the procedure has not been written carefully enough, and it is giving an unintended result. In the example above, the programmer probably did not consider the case that the temporary variable was the same as one of those being interchanged, and neither provided for it, nor warned potential users. A comment of the form

```
{Actual parameters must be distinct identifiers}
```

would have been helpful.

When there is no aliasing it is safe to calculate the procedure's part function in isolation and substitute the actual parameters in the result; when aliasing occurs, however, the definition must be used directly, copying in the actual parameter before the part-function calculation.

9.1.5 Exercises

9.1.1 Create procedures with parameters for:

a) `SelectSort` (Section 4.2.1),

b) `SelectReverse` (Section 4.2.4),

c) Do a) and b) together with a single procedure, using a parameter to choose sorting or reversing,

d) `BubbleSort` (Section 4.3.4).

Be careful to precisely define the state that files must be in before your procedures are invoked, and after their execution.

9.1.2 Write a test-driver program for sorting procedures like those of a), b), d) in the previous exercise. The file to be sorted comes from the test-driver input, and the result goes to its output. Give an input file which when used in this way will force a complete structural test of `SelectSort`.

9.1.3 For the version of `Split` described in Section 9.1.4:

a) Write out the complete program.

b) Write out the part function for `CopyOut` as a conditional assignment (that is, the part function of a procedure statement in which `CopyOut` has the same actual parameters as its formal parameters).

c) Is there any way aliasing can occur with `CopyOut`?

d) Write out the complete meaning for `Split` as a conditional assignment.

9.1.4 Consider a procedure that sometimes prints values of a variable `MonitoredV`:

```
PROCEDURE BugOut;
  BEGIN
    IF Buggy = 'Y'
    THEN
      IF MonitoredV <> HoldV
      THEN
        BEGIN
          HoldV := MonitoredV;
          WRITELN('Changed value is: ', HoldV)
        END
  END
```

a) What declarations and initializations are needed to go with this procedure?

b) Explain in English how the procedure might be used for debugging.

c) Work out the part function of the procedure statement; that is, calculate `BugOut`.

9.1.5 A parametrized version of `BugOut` (previous exercise) can be written to deal with any variable.

a) Write `BugOut` with parameters to that it is maximally useful, but self-contained. Observe the all-or-nothing rule.

b) Initializations will still be required with the modified procedure, so write another procedure `StartBug` to initialize for use of `BugOut`. Write comments on both procedures that explain how to use them, so that users need not read their BEGIN statements.

9.1.6 The all-or-nothing rule suggests inside a procedure avoiding the use of program variables declared outside. Explain why actual parameters are really declared outside, but why this doesn't violate the spirit of the rule.

9.1.7 Often a procedure would like to distinguish between the first call on it, and all subsequent calls.

a) Explain how to do this using a variable declared outside the procedure.

b) Explain how to do it using a parameter.

c) Which way seems better, and why?

9.1.8 Write a procedure `NextDigit` with one parameter that advances the value of the parameter around the cycle **01234567890**.

9.1.9 Some parameters are intended for input to a procedure, others for output from the procedure, and occasionally for both. These parameters are for communication with the rest of the program. But sometimes parameters only serve as a kind of "memory" for the procedure itself, as in

```
PROCEDURE ManyCalls(VAR First, Last: CHAR);
  BEGIN
    IF First = 'Y'
    THEN
      BEGIN
        Last := '0';
        First := 'N'
      END
    ELSE
      NextDigit(Last); {see previous exercise}
    WRITELN('This is call ', Last)
  END
```

Assuming that the actual parameters supplied for `First` and `Last` have no other usage, rewrite the procedure with a single parameter to give the same effect. Include comments that explain clearly how the procedure is to be used.

9.1.10 Aliasing can involve several parameters instead of a parameter and a globally declared identifier as in the text. Analyze the situation in the example:

```
PROGRAM AliasToo(INPUT,OUTPUT);
  PROCEDURE MessUp(FilV: TEXT);
    BEGIN {MessUp}
      WRITE('A');
      WRITE(FilV, 'B')
    END;  {MessUp}
  BEGIN {AliasToo}
    MessUp(OUTPUT);
  END.  {AliasToo}
```

9.1.11 Formulate a rule that can insure the *absence* of aliasing, so that a procedure statement's meaning can be obtained by substituting actual identifiers for formal ones in the meaning of the procedure body. (Hint: the relevant ideas are variables not local to the body, and actual parameters.)

9.1.12 For the procedure `Copy` (Section 9.1.1),

a) Calculate the meaning of the procedure statement:

```
Copy(InF, OutF, Temp)
```

b) If aliasing occurs in calling `Copy`, does anything unexpected happen? How is this reflected in the formal definition?

9.1.13 Calculate `Compare(First, Second, YesNo)`, for the declaration:

```
PROCEDURE Compare(VAR Ch1, Ch2, Result: CHAR);
  BEGIN
    IF Ch1 = Ch2
    THEN
      Result := 'Y'
    ELSE
      Result := 'N'
  END
```

Does aliasing occur in this call? Can aliasing occur in *any* call? Explain.

9.2 Procedures with Declarations

Preview

In addition to the variable names that are the parameters of a procedure, local variables can be declared for use only within the procedure. These variables can hide others declared outside the procedure. The existence of variables at different positions in a program raises the topic of variable scope. To illustrate the use of local variables, a basic library of string-comparison operations is given.

New ideas: declaring local variables in a procedure, variable scope, resolving variable-name conflicts, string comparison, lexicographic ordering.

Parameters are used, as well as other program variables, in statements within the procedure. However, during execution formal parameters are replaced by corresponding actual parameters, program variables that exist outside the procedure. Therefore, in effect, procedure executions are carried out entirely using variables declared in the calling program.

Sometimes, however, variables used in a procedure execution should be simply temporary or working variables, whose values never involve the calling program. For example, the procedure to exchange the values of two variables using a third temporary variable, has been written as

```
PROCEDURE SwitchThese2(VAR V1, V2, Temp: CHAR);
  BEGIN
    Temp := V1;
    V1 := V2;
    V2 := Temp
  END
```

with `Temp` as a parameter. Then, in order to exchange the values of two variables, say `One` and `Two`, the using program will need to supply an additional variable to serve as the temporary, e.g.,

```
SwitchThese2(One, Two, Ch)
```

which will exchange the values of `Two` and `One`, and also set `Ch` to the old value of `One`.

The ability to declare local variables in CF Pascal allows variables like `Temp` to be hidden within the procedure that uses them. For example,

```
PROCEDURE Exchange(VAR Ch1, Ch2: CHAR);
  {Exchange Ch1 and Ch2}
  VAR
    Temp: CHAR;
  BEGIN
    Temp := Ch1;
    Ch1 := Ch2;
    Ch2 := Temp
  END
```

does the exchange in this way. Like the formal parameters `Ch1` and `Ch2`, the local variable `Temp` is not known to, and cannot be referenced by, statements outside the procedure `Exchange`. The procedure statement

```
Exchange(One, Two)
```

will exchange the values of `One`, `Two`, and `Exchange` can be described for external use and reuse more simply than `SwitchThese2`, since it uses and affects only two variables instead of three.

Local variables have a fleeting existence. They enter the execution state at the beginning of execution of a procedure statement (just as a program's variables enter the state at the beginning of its execution), and they disappear from the state when the procedure-statement execution is completed. Execution tables show this behavior just as they did for the variables of programs. For example, the program:

```
PROGRAM Copy2(INPUT,OUTPUT);
  {Copy to the first two characters from INPUT to OUTPUT}
  PROCEDURE CopyChar(VAR F1, F2: TEXT);
    {Copy to F2 the next character from F1}
    VAR
      Ch: CHAR;
    BEGIN {CopyChar}
      IF NOT EOF(F1)
      THEN
        BEGIN
          READ(F1, Ch);
          WRITE(F2, Ch)
        END
    END;  {CopyChar}
  BEGIN {Copy2}
    CopyChar(INPUT, OUTPUT);
    CopyChar(INPUT, OUTPUT);
    WRITELN
  END.  {Copy2}
```

Execution

```
INPUT :ABC
OUTPUT:AB
```

has the execution table:

After execution of	INPUT	OUTPUT	Ch	EOF
	ABC			
PROGRAM Copy2(INPUT,OUTPUT)	**ABC/**	_		FALSE
BEGIN {Copy2}				
CopyChar(INPUT, OUTPUT)				
VAR Ch: CHAR			?	
BEGIN				
IF NOT EOF(F1)				FALSE
READ (INPUT, Ch)	**ABC/**		**A**	
WRITE (OUTPUT, Ch)		**A**_		
END			§	
CopyChar(INPUT, OUTPUT)				
VAR Ch: CHAR			?	
BEGIN				
IF NOT EOF(F1)				FALSE
READ (INPUT, Ch)	**ABC/**		**B**	
WRITE (OUTPUT, Ch)		**AB**_		
END			§	
WRITELN		**AB/**_		
END. {Copy2}	**ABC**	**AB**		

The local variable Ch ceases to exist in the state at the places marked § in this table. Thus even though the two procedure statements invoking CopyChar are together, the value of Ch is lost, indeed the variable ceases to exist, between them.

Within a procedure there may be local procedure declarations as well as local variable declarations. Local procedures may be invoked only within the procedure in which they are declared, and are entirely invisible outside it. The value that enters and leaves the execution state for a local procedure is its text. The entire business of declarations within procedures is exactly like declarations in a program, except that identifiers may only be referenced within the procedure, and enter/leave the execution state at each procedure statement.

9.2.1 Identically Named Variables

In a large program, the procedures may be developed concurrently by different people. If every local variable had to have a unique name, this kind of parallel development would be difficult, since variable names would have to be assigned by some central authority. Furthermore, a requirement for unique local names would make it difficult to reuse a procedure, since its names might conflict with the new application. Fortunately, Pascal allows local names to be used over and over. This is in accord with the idea that local identifiers have no existence outside the procedure in which they are declared, and so there should be no reason not to reuse them. It is more surprising that Pascal allows a local identifier to override one that exists in the surrounding context. Variables that could be used in a procedure, but are not declared there, are called *global*.

Consider the following program in which there is a global variable Ch, and also a local Ch, whose values exist simultaneously, even though only one is accessible at any time:

```
PROGRAM CopyR(INPUT,OUTPUT);
  {Copy to OUTPUT, in reverse order, the
   first two characters from INPUT}
  VAR
    Ch: CHAR;
  PROCEDURE CopyChar(VAR F1, F2: TEXT);
    {Copy to F2 the next character from F1}
    VAR
      Ch: CHAR;
    BEGIN {CopyChar}
      IF NOT EOF(F1)
      THEN
        BEGIN
          READ(F1, Ch);
          WRITE(F2, Ch)
        END
    END;  {CopyChar}
  BEGIN {CopyR}
    READ(Ch);
    CopyChar(INPUT, OUTPUT);
    WRITELN(Ch)
  END. {CopyR}
```

Execution

```
INPUT :ABC
OUTPUT:BA
```

The READ statement of CopyR:

```
READ(Ch)
```

references the global Ch declared in CopyR. However, during the execution of the procedure statement

```
CopyChar(INPUT, OUTPUT)
```

the local Ch (declared in CopyChar) supersedes the global, and this local Ch is referred to in the READ, WRITE statements of the procedure:

```
READ (F1, Ch);
WRITE (F1, Ch)
```

During the time that the local Ch supersedes the global Ch, the global is not lost; no reference to it is possible, but its value is maintained by the Pascal machine. Then when the execution of CopyChar is completed, the local Ch disappears, the global is again available, with its value intact. Thus, the final WRITE statement:

```
WRITE(OUTPUT, Ch)
```

refers to the global Ch. The execution table below traces the values of both Ch variables. The symbol § is used to explicitly mark a variable that does not exist. The region between the horizontal lines in the global Ch column is the portion of execution during which this variable is inaccessible.

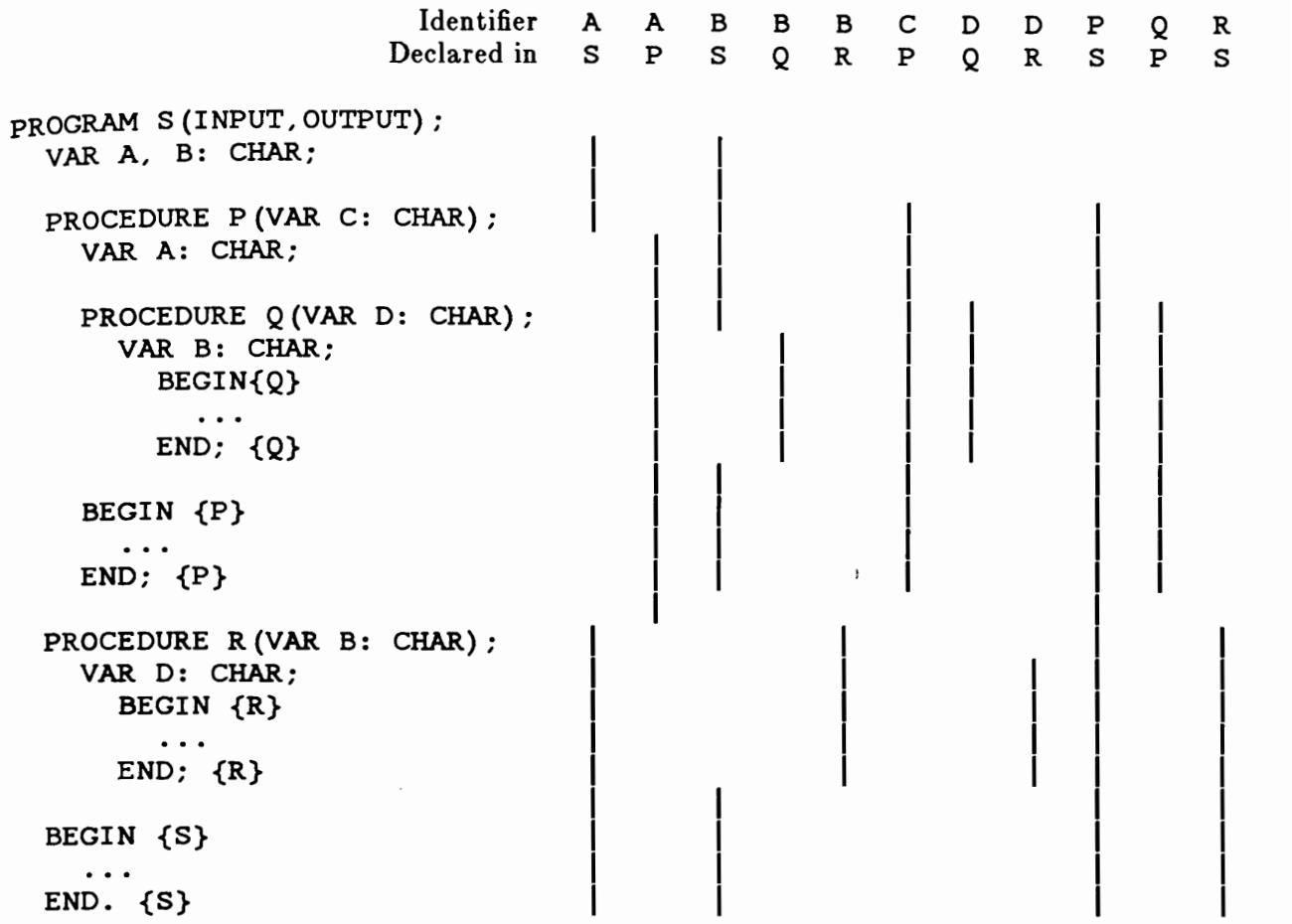
After execution of	INPUT	OUTPUT	global Ch	local Ch
	ABC			
PROGRAM CopyR(INPUT,OUTPUT)	**ABC**/	_	§	§
VAR Ch: CHAR			?	
BEGIN {CopyR}				
READ(Ch)	**A**B**C**/		**A**	
CopyChar(INPUT, OUTPUT)				
VAR Ch: CHAR				?
BEGIN				
READ(INPUT, Ch)	**AB**C/	_		**B**
WRITE(OUTPUT, Ch)		**B**_		
END				§
WRITELN(Ch)		**BA**/_		
END. {Copy1}	**ABC**	**BA**		

Thus, in this execution, **A** is read first, but written last, after **B** is read and written by CopyChar. At the second declaration of Ch the value of global Ch is hidden, and it reappears when the local Ch disappears.

9.2.2 Scope of Identifiers

Declarations within procedures, possibly of other procedures that in turn may contain declarations, raise tricky questions about which identifiers can be used at each point in a program. The *scope* of an identifier is the set of statements in which the identifier can be referenced. In Pascal an identifier can only be referenced within the <block> containing its declaration. The scope of an identifier is at most the BEGIN statements nested within the PROCEDURE or PROGRAM containing the identifier's declaration, (i.e., the BEGIN statement and any nested procedures' BEGIN statements). However, the scope can be smaller than this, if there is a nested block within which the same identifier is declared. Then the global scope has a hole in it, the inner block, in which the conflicting local has sway.

Most of the complications of identifier scope can be understood by mastering an example. In the figure below, the scope of each identifier is shown by a vertical line running beside those statements in which it can be referenced. To distinguish the same identifier declared in different places, the name of the procedure/program is used.



Notice that P can be referenced in P, Q, R, and S because it is declared in S, all these procedures are contained in S, and none of the procedures contains a new declaration for P. R doesn't have the same scope as P only because of its later declaration: it is not visible in P or Q because Pascal insists that identifiers be declared before they are used. In Q, the declarations of A and B in S are hidden by a declaration in P and Q, respectively. The B in S is visible in P outside the definitions of Q and R.

Because Pascal's rules of scope depend on the structure of the text of a program rather than its execution behavior, they are called *static scope* rules. The scope of an identifier is determined once and for all by the syntax of the program in which it appears, and all questions about scope can be answered by examining the program text.

9.2.3 Syntax and Context Rules

Because the form of a procedure is exactly like that of a program, the syntax quantity <block> describes the declaration and statement structure, so it is only necessary to change rule 27:

SR27. <procedure> ::= <procedure heading> ; <block>

to accommodate the additional forms.

The discussion of scope permits a simple context rule:

CR7. An identifier may not be used outside its scope.

9.2.4 String Manipulation Procedures

To illustrate the use of local variable declarations, two useful string-comparison procedures will be developed. These procedures have two file input parameters for the strings, and a character result parameter. The remainder of the working storage is internal to the procedures.

Consider the problem of comparing the lengths of strings stored in distinct files `F1` and `F2`. The result of the comparison is returned in `Result`: **1** if the string in `F1` is shorter, **2** if the string in `F2` is shorter, and **0** if they are of equal length.

Design Part 1

```
PROCEDURE Length(VAR F1, F2: TEXT; VAR Result: CHAR);
  {Result is 0, 1, 2 if length of F1 =, <, >
   length of F2 respectively.  The actual parameters
   corresponding to F1 and F2 must be distinct.}
  VAR
    Ch1, Ch2: CHAR;
  BEGIN {Length}
    RESET(F1);
    RESET(F2);
    WHILE NOT EOF(F1) AND NOT EOF(F2)
    DO
      BEGIN
        READ(F1, Ch1);
        READ(F2, Ch2)
      END;
    {Analyze F1, F2 and set Result for length}
  END {Length}
```

Design Part 1.1

```
BEGIN {Analyze F1, F2 and set Result for length}
  IF EOF(F1)
  THEN {F1 is empty}
    IF EOF(F2)
    THEN {F1 and F2 are both empty}
      Result := '0'
    ELSE { F1 is empty but F2 has more characters}
      Result := '2'
  ELSE {F1 has more characters, but F2 is empty}
      Result := '1'
END
```

The *lexicographic* order of strings is their order of appearance as words in a dictionary—their alphabetic order, where the alphabet is the character set of the Pascal machine. If two strings of different lengths agree throughout the shorter one, the shorter precedes the longer. For example, †**ABC**† precedes †**ABCDEF**† but †**XYZ**† precedes †**YZ**†.

It is a simple matter to determine the lexicographic order of two strings—compare their characters pairwise until a nonequal pair or the end of one (or both) strings is found. Although `Lexico` is meant to be used with strings on a single line, it is robust enough to be used with strings that cross line boundaries.

Design Part 2

```
PROCEDURE Lexico(VAR F1, F2: TEXT; VAR Result: CHAR);
  {Result is 0, 1, 2 if lexicographic order of
   F1 =, <, > that of F2.  The actual parameters
   corresponding to F1 and F2 must be distinct.}
  VAR
    Ch1, Ch2: CHAR;
  BEGIN {Lexico}
    RESET(F1);
    RESET(F2);
    Result := '0';
    WHILE (NOT EOF(F1)) AND (NOT EOF(F2)) AND (Result = '0')
    DO
      BEGIN
        READ(F1, Ch1);
        READ(F2, Ch2);
        IF (Ch1 < Ch2) OR (EOF(F1))
        THEN {Ch1 < Ch2 or F1 shorter than F2}
          Result := '1'
        ELSE
          IF (Ch1 > Ch2) OR (EOF(F1))
          THEN {Ch1 > Ch2 or F2 shorter than F1}
            Result := '2'
      END
  END {Lexico}
```

Length and Lexico are useful building blocks in solving problems involving strings. The programs SelectSort and SelectReverse developed in Section 4.2 can also be converted into procedures. This nucleus can be extended to form a library of routines for manipulating strings. The proper routines to have in a library is a matter for personal taste and experience. As the library is used, and as more is learned about string algorithms, it can be expected to evolve into a powerful aid to solving problems.

When library routines are to be used in a design, it is convenient to give their names and a summary comment, but not the full text. The comment should be enough to correctly employ each procedure, and the full text will be inserted (perhaps by some automatic system) when the design is complete. The standard form used in this text is an *include comment*. For example, if Lexico were needed in a design, the comment would be:

```
{Include PROCEDURE Lexico
   (VAR F1, F2: TEXT; VAR Result: CHAR)
     Result is 0, 1, 2 if lexicographic order of
     F1 =, <, > that of F2.  The actual parameters
     corresponding to F1 and F2 must be distinct.}
```

Thus each procedure in the library must begin with the information needed to construct the include comment. The procedure heading supplies the name and detailed information about the parameters. The remainder of the include comment comes from documentation that allows the use of a procedure without examining its text; in particular, it must note restrictions and special cases carefully.

Unless otherwise noted, the strings in parameter files contain no line breaks; input files will be initially RESET by the procedures, but will not necessarily be left in the RESET state; output files will have an initial REWRITE and will be left open for writing; aliasing of actual parameters is not allowed.

```
PROCEDURE InString(VAR F1: TEXT)
  {Create the string in F1 from the characters
   in INPUT, up to the next line break}
PROCEDURE OutString(VAR F1: TEXT)
  {Print the string in F1 as a separate line of output}
PROCEDURE CopyOpen(VAR F1, F2: TEXT)
  {Copy string in F1 into F2 without RESET or REWRITE;
   thus F1 must be ready to read and F2 ready to write,
   but the past strings of these files may not be empty}
PROCEDURE Sort(VAR F1, F2: TEXT)
  {F2 is characters of string F1 sorted}
PROCEDURE Reverse(VAR F1, F2: TEXT)
  {F2 is characters of string F1 in reverse}
PROCEDURE Concatenate(VAR F1, F2, F3: TEXT)
  {Create F3 as the contents of F1 concatenated
   with the contents of F2}
```

Library routines are often used in writing other library routines. For example, the design for Concatenate might be:

Design Part 3

```
PROCEDURE Concatenate(VAR F1, F2, F3: TEXT);
  {Create F3 as the contents of F1 concatenated
   with the contents of F2}
  {Include PROCEDURE CopyOpen(VAR F1, F2: TEXT)
    Copy string in F1 into F2 without RESET or REWRITE;
    thus F1 must be ready to read and F2 ready to write,
    but the past strings of these files may not be empty}
  BEGIN
    RESET(F1);
    RESET(F2);
    REWRITE(F3);
    CopyOpen(F1, F3);
    CopyOpen(F2, F3)
  END
```

9.2.5 Identifier Confusion

In most cases no additional definitions are needed to use the program calculus on CF Pascal procedures with local variables. Local variables occur within the <block> of the declaration, which is attached as value to the procedure identifier in the state. The meaning of a procedure statement is obtained from this value, and the local variables will enter the state through $\boxed{\texttt{VAR ...}}$, and leave it through $\boxed{\texttt{VAR ...}}^{\mathrm{T}}$, just as global variables do. Consider, for example, the program:

```
PROGRAM NoCon(INPUT,OUTPUT);
  VAR
    Global: CHAR;
  PROCEDURE HasLocal;
    VAR
      Local: CHAR;
    BEGIN {HasLocal}
      Local := 'L'
    END;  {HasLocal}
  BEGIN {NoCon}
    Global := 'G';
    HasLocal
  END.  {NoCon}
```

The execution state just before the procedure statement is

$$S = \{\texttt{INPUT}\cdot\langle x,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Global}\cdot\mathbf{G}\}$$

where x is the input string, and

$$\boxed{\texttt{HasLocal}}(S) = \boxed{S(\texttt{HasLocal})}(S)$$

$$= \boxed{\texttt{VAR Local: CHAR; BEGIN Local := 'L' END}}(S)$$

$$= \boxed{\texttt{VAR Local: CHAR}}^{\mathrm{T}}\left(\boxed{\texttt{Local := 'L'}}\left(\boxed{\texttt{VAR Local: CHAR}}(S)\right)\right)$$

$$= \boxed{\texttt{VAR Local: CHAR}}^{\mathrm{T}}\left(\boxed{\texttt{Local := 'L'}}\right.$$
$$\left.(\{\texttt{INPUT}\cdot\langle x,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Global}\cdot\mathbf{G},\ \texttt{Local}\cdot?\})\right)$$

$$= \boxed{\texttt{VAR Local: CHAR}}^{\mathrm{T}}$$
$$(\{\texttt{INPUT}\cdot\langle x,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Global}\cdot\mathbf{G},\ \texttt{Local}\cdot\mathbf{L}\})$$

$$= \{\texttt{INPUT}\cdot\langle x,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Global}\cdot\mathbf{G}\}$$

which has exactly the proper effect: the local variable enters the execution state, is present for the statements within the procedure to act on, then disappears.

Unfortunately, the definition goes wrong if it is blindly applied to programs in which there is duplication of local and global identifiers. Consider what would happen in calculating the meaning of:

```
PROGRAM Confuse(INPUT,OUTPUT);
  VAR
    MixedUp: CHAR;
  PROCEDURE HasLocal;
    VAR
      MixedUp: CHAR;
    BEGIN {HasLocal}
      MixedUp := 'L'
    END;  {HasLocal}
  BEGIN {Confuse}
    MixedUp := 'G';
    HasLocal
  END.  {Confuse}
```

This program ought to have exactly the same meaning as the first one, since the scopes of the two variable identifiers do not permit them to be simultaneously available: the local has the **L** assignment, and the global the **G** assignment. But applied literally, the definition for the procedure statement would add to a state containing the pair

$\langle$MixedUp, **G**$\rangle$

(for the global variable), all other pairs

```
<MixedUp, A>
<MixedUp, B>
  ...
```

for the local variable. This is evidently wrong; there has been a literal confusion of the identifiers that should have been kept distinct.

The difficulty is precisely that local identifiers are not supposed to have any connection with those that already exist in an execution state to which the local `VAR...` part function is applied. And the strictly transitory nature of a local identifier supplies a way out of the difficulty: it can be systematically replaced without altering the meaning of the procedure, because it is a dummy variable.

The meaning of a procedure statement in which local declarations conflict with global ones therefore requires systematic replacement of the local identifier in the procedure declaration's text, using an identifier that does not already appear in the execution state. One way to create such a new identifier is to append the character `N` to the one in conflict, and if that is still in conflict, append `N` again, etc. Eventually an identifier not in the execution state will be created; and by using it throughout the procedure declaration, the confusion will be avoided.

The formal definition of the part function of a procedure statement for the declaration

`PROCEDURE` P `(VAR` X `: CHAR); VAR` Z `: CHAR; BEGIN ... END`

thus becomes:

$$\boxed{P\ (\ Y\ )}(s) = \boxed{s\,(P)_{X \leftarrow Y,\, Z \leftarrow Z'}}(s)$$

when $Z \in \text{domain}(s)$, where Z' is a new identifier.

For the example above, using the scheme of appending the character `N` to make a new identifier, the result is

$$\boxed{\texttt{HasLocal}}(S) = \boxed{S(\texttt{HasLocal})}(S)$$

$$= \boxed{\texttt{VAR MixedUpN: CHAR; BEGIN MixedUpN := 'L' END}}(S)$$

and now everything works as it should.

Technically, it is not necessary to examine the execution state to determine that an identifier substitution must be made, nor how many `N` characters must be appended. Because of Pascal's static scope rules, all potential conflicts can be deduced from the program's structure, and a conflict-free program could be written in which no substitutions are necessary.

Here is an example program called W in which confusion of identifiers could lead to doubt about what the program actually does.

```
PROGRAM WhichWhich(INPUT,OUTPUT);
  VAR
    Which: CHAR;
  PROCEDURE Zap;
    BEGIN {Zap}
      Which := 'Z' {Zap it, but which?}
    END;  {Zap}
  PROCEDURE Invoke;
    VAR
      Which: CHAR;
    BEGIN {Invoke}
      Which := 'B';
      Zap;
      WRITE(Which)
    END;  {Invoke}
  BEGIN  {WhichWhich}
    Which := 'A';
    Invoke;
    WRITELN(Which)
```

```
END.    {WhichWhich}
```

The variables used in the WRITE statements are clear; the local to `Invoke` is used in the WRITE there, and the global in the outer WRITELN. However, between the assignments and the WRITE statements, one of the variables gets `Zapped`. If the local is overwritten, then the program's meaning is the constant function

$$\boxed{W} = \langle\dagger\mathbf{ZA}\dagger\rangle$$

or, if the global is changed, the meaning is

$$\boxed{W} = \langle\dagger\mathbf{BZ}\dagger\rangle.$$

In difficult cases like this one, there is a way to guess the meaning without calculating the program function. Imagine changing one of the `Which` identifiers systematically. If the local were changed the program would still be syntactically correct, and the output would be †**BZ**†. If the global were changed, the program would not be syntactically correct, since the reference to `Which` in `Zap` would be to an undeclared variable. Hence it ought to be the global variable that gets Zapped. Formal definitions are useful in deciding difficult cases, however.

Using the modified definition on the execution state

$$S = \{\texttt{INPUT}\cdot\langle x,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Which}\cdot\mathbf{A}\}$$

that exists at the outer procedure statement,

$$\boxed{\texttt{Invoke}}(S) = \boxed{S(\texttt{Invoke})}(S)$$

$$= \boxed{\texttt{VAR WhichN: CHAR; BEGIN WhichN := 'B'; ... END}}(S)$$

$$= \boxed{\texttt{VAR WhichN: CHAR}}^{\mathrm{T}}\Big(\boxed{\texttt{BEGIN WhichN := 'B'; ... END}}\ \big(\boxed{\texttt{VAR WhichN: CHAR}}(S)\big)\Big)$$

$$= \boxed{\texttt{VAR WhichN: CHAR}}^{\mathrm{T}}\Big(\boxed{\texttt{BEGIN WhichN := 'B'; ... END}}\ (\{\texttt{INPUT}\cdot\langle x,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Which}\cdot\mathbf{A},\ \texttt{WhichN}\cdot?\})\Big)$$

$$= \boxed{\texttt{VAR WhichN: CHAR}}^{\mathrm{T}}\Big(\boxed{\texttt{WhichN := 'B'; Zap; WRITE(Which)}}\ (\{\texttt{INPUT}\cdot\langle x,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Which}\cdot\mathbf{A},\ \texttt{WhichN}\cdot?\})\Big)$$

$$= \boxed{\texttt{VAR WhichN: CHAR}}^{\mathrm{T}}\Big(\boxed{\texttt{Zap; WRITE(Which)}}\ (\{\texttt{INPUT}\cdot\langle x,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Which}\cdot\mathbf{A},\ \texttt{WhichN}\cdot\mathbf{B}\})\Big)$$

$$= \boxed{\texttt{VAR WhichN: CHAR}}^{\mathrm{T}}\Big([\text{WRITE(Which)}\ (\{\texttt{INPUT}\cdot\langle x,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Which}\cdot\mathbf{Z},\ \texttt{WhichN}\cdot\mathbf{B}\})\Big)$$

$$= \boxed{\texttt{VAR WhichN: CHAR}}^{\mathrm{T}}\ (\{\texttt{INPUT}\cdot\langle x,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\mathbf{B}\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Which}\cdot\mathbf{Z},\ \texttt{WhichN}\cdot\mathbf{B}\})$$

$$= \{\texttt{INPUT}\cdot\langle x,\dagger\dagger,\mathbf{R}\rangle,\ \texttt{OUTPUT}\cdot\langle\dagger\mathbf{B}\dagger,\dagger\dagger,\mathbf{W}\rangle,\ \texttt{Which}\cdot\mathbf{Z}\}$$

From this point it is easy to calculate that

$$\boxed{W} = \langle\dagger\mathbf{BZ}\dagger\rangle.$$

Although it is always possible to write programs in which there are no possible confusions between local and global identifiers, CF Pascal allows this confusion. It can be eliminated by systematically changing the names of the local identifiers, whose existence is in any case fleeting.

9.2.6 Exercises

9.2.1 Consider the problem of aliasing raised in `SwitchThese2` (Section 9.1.4).

a) Explain why it is impossible for the programmer to allow arbitrary aliasing and still get the intended results, so long as local variables are not used.

b) Rewrite the procedure so that it gets proper results no matter what aliasing occurs.

9.2.2 When a parameter is used only for "memory" in a procedure, as is `Last` in `ManyCalls` of Exercise 9.1.9, why can't it be made local to the procedure that needs the memory?

9.2.3 Suppose that several procedures, written independently by different people, are to be combined into a single program. A procedure may contain: (1) references to variables not declared within it, (2) parameters, and (3) local variables. For each case (1)-(3) explain why difficulty can or cannot arise because of a chance use of the same identifier by the different programmers. If there is a difficulty, give an example to illustrate it.

9.2.4 If a program uses only *one* identifier (say `Unique`) in all of its `VAR` declarations (in the main block, procedure blocks, parameters, etc.), can it make use of more than one *value* simultaneously? Explain why or why not. Can it *retain* more than one value at a time? Explain.

9.2.5 Can a procedure's name, its parameter names, or its local variable names use exactly the same identifier? Can each of these be the same identifier as a variable declared external to the procedure? Explain.

9.2.6 Give an example in which a variable's scope has a "hole" in it; that is, exhibit a syntactically contiguous part of a program that starts within a scope and ends there, but the scope does not cover it all.

9.2.7 Formulate a rule that elaborates on the "all or nothing" rule of Section 9.1.3, to help a programmer decide when to use a variable declared outside a procedure, when to use a parameter, and when to use a local variable. The intent of your rule should be to make procedures as self-contained as possible, and to eliminate problems with aliasing.

9.2.8 Make up a set of structural tests for `Length` and `Lexico` of Section 9.2.4.

9.2.9 For `Length` (Section 9.2.4):

a) Rewrite the summary comment as a conditional assignment, making it as complete and precise as possible.

b) Calculate the meaning of the procedure, in terms of its parameters. Can anything unintended occur if there is aliasing with the parameters of the procedure? If so, give an example of unexpected behavior; if not, explain why not.

9.2.10 Rewrite the comment for each of the procedures in the starter library of Section 9.2.4 as a conditional assignment, making it as complete as possible.

9.2.11 The restrictions on parameters in the string-processing routines forbid aliasing.

a) Give an argument why such restrictions are inappropriate for library routines.

b) How could the routines be written so that the aliasing restriction was removed? (Consider `Concatenate` as an example.)

9.2.12 If the original definition were blindly applied to the example `Confuse` in Section 9.2.5, what would the execution state be just after the procedure statement? Would `MixedUp` be present?

9.2.13 Explain why, when a program is being written, identifier confusion like that shown in the program `WhichWhich` of Section 9.2.5 can be avoided by simply keeping a list of variables, and not reusing them—that is, show that programmers are only accidentally confused in this way.

9.2.14 Can it happen that the declaration to which a certain identifier refers changes during program execution? That is, for a given program and one particular occurrence of an identifier *V* within it, could there be one execution where this occurrence of *V* refers to one declaration, and yet another execution where it refers to a different declaration? Explain, and comment on whether this aspect of Pascal is good the way it is.

9.3 Processing Natural Numbers

Preview

Natural numbers can be represented as character strings. This section demonstrates how to perform arithmetic on such representations. The procedures created form a useful library.

New ideas: natural numbers as character strings, arithmetic on strings.

9.3.1 The Natural Numbers

Everyone learns the counting numbers zero, one, two, three, four, five, ..., as a child. The mathematical name for this collection is the *natural numbers*. "Natural" does not mean originating in nature, but means in common use. Usually we learn the natural numbers from our parents, who learned them from their parents, and so on. But going back far enough, nobody's parents knew the natural numbers. They are human artifacts, just as language, the plays of Shakespeare, the Constitution of the United States, are.

Natural numbers can be studied as an endless series of mathematical objects such that, given any one of them, the next object is always defined. In order to get started, one assumes there is a distinguished starting object, which describes all empty collections, called zero, or 0. Then

next after 0 is 1;
next after 1 is 2;
next after 2 is 3;

and so on, where the symbols are conventional names for these "next after" objects. These conventional names might be thought of as unique digit strings. In particular, the strings

†0†, †1†, †2†, ..., †10†, †11†, ..., †100†, ...

are familiar names of natural numbers. These representations are given in place notation, in base ten. Place notation has been widely known for little more than a thousand years. It is so simple and elegant that it eluded the best minds of the ancient world. Euclid did not know the place notation, even though he knew many deep things about the natural numbers (for example, that there is no largest prime, and how to find the greatest common divisor of two natural numbers).

Natural numbers can be represented in place notation in bases other than ten:

Base	Natural Numbers							
two	0	1	10	11	100	101	110	...
three	0	1	2	10	11	12	20	...
four	0	1	2	3	10	11	12	...
...								
ten	0	1	2	3	4	5	6	...

In base b, digits 0, 1 ..., b-1 are used, and the number b in base b is always written 10, b^2 is written 100, etc. To tell whether 11 is three (base two), four (base three), ..., eleven (base ten), etc., the base must be known. Unless specified otherwise, the base will be ten, and when the numbers are written out in English the emphasis is put on the number itself (as two is the idea of all pairs, while 10 is merely the base-two representation of this idea, for example). It is dangerous to use the phrase "base 10" in place of "base ten". Every representation in any base b is "base 10" because b is always written 10!

Arithmetic is what natural numbers do best. Two natural numbers can be (for example) multiplied to get another natural number. There are numbers for which multiplication is not useful—for example, license plate numbers or telephone numbers. On the other hand, dates are numbers, and while multiplication of dates isn't of interest, subtraction can be: the difference between two dates is the elapsed time from the later to the earlier.

9.3.2 One-Digit Addition

Every school child knows the base-ten addition table, but CF Pascal does not. Therefore, to write programs for addition in general, one way is to begin with a procedure for one-digit addition. The addition table has rows and columns labeled with the digits 0...9. At each intersection of a row and column is the sum of the row and column digits:

One-digit Addition Table

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

Let the one-digit addition procedure be `AddDigits` with the procedure heading:

```
PROCEDURE AddDigits(VAR D1, D2, S1, S2: CHAR);
  {Add digits D1, D2 to get sum as digits S1, S2
   where S1 is 0 for a one-digit sum in S2, or 1
   for a two-digit sum whose unit digit is S2.}
```

The procedure could be written directly in a single deeply nested IF statement by selecting the row defined by `D1`, column defined by `D2`, then returning the proper `S1`, `S2` values given in the table. Such a procedure would have hundreds of lines of text, but it would work.

The regularity of the addition table suggests that there is a shorter way to write `AddDigits`. The brute-force approach makes no use of this regularity. The units digit of a sum is particularly regular, as can be seen by giving it a table of its own:

Unit's Digit Addition Table

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

Across each row, the digits increase in natural-number order, if digit 0 is considered to be next after digit 9. Thus the last digit in (say) the row for 7, column for 5, can be obtained by counting to 5 starting with 7: 8,9,0,1,2. And the first digit of the result is 1 just in case this count has gone through 0. The design for `AddDigits` uses these ideas, with the counting done by a procedure `NextDigit`:

Design Part 4

```
PROCEDURE AddDigits(VAR D1, D2, S1, S2: CHAR);
  {Add digits D1, D2 to get sum as digits S1, S2
   where S1 is 0 for a one-digit sum in S2, or 1
   for a two-digit sum whose unit digit is S2.}
  VAR
    Count: CHAR;
  PROCEDURE NextDigit(VAR D1: CHAR);
    {D1 is replaced by its successor in the
     sequence 0123456789012...}
  BEGIN {AddDigits}
    S1 := '0';
    S2 := D1;
    Count := '0';
    WHILE Count < D2
    DO
      BEGIN
        NextDigit(S2);
        IF S2 = '0'
        THEN
          S1 := '1';
        NextDigit(Count)
      END
  END {AddDigits}
```

Design Part 4.1

```
PROCEDURE NextDigit(VAR D1: CHAR);
  {D1 is replaced by its successor in the
   sequence 0123456789012...}
  BEGIN {NextDigit}
    IF D1 = '0' THEN D1 := '1' ELSE
    IF D1 = '1' THEN D1 := '2' ELSE
    IF D1 = '2' THEN D1 := '3' ELSE
    IF D1 = '3' THEN D1 := '4' ELSE
    IF D1 = '4' THEN D1 := '5' ELSE
    IF D1 = '5' THEN D1 := '6' ELSE
    IF D1 = '6' THEN D1 := '7' ELSE
    IF D1 = '7' THEN D1 := '8' ELSE
    IF D1 = '8' THEN D1 := '9' ELSE
    IF D1 = '9' THEN D1 := '0'
  END  {NextDigit}
```

This procedure has about 30 lines of noncomment text—some improvement over the first try.

To test `AddDigits` structurally, first consider the procedure `NextDigit`. It must be given each different digit for all statements to be used. This can be arranged by taking the `D2` value to be **9** so there will be nine iterations, with the actual parameter `Count` to `NextDigit` taking values **0**, **1**, **2**, ..., **8**. If the parameter `S2` takes the value **9**, which will happen for any `D1` value except **0**, the missing case of **9** will be tried. Thus the values **9** for `D2`, **3** for `D1` will structurally test `NextDigit`. This case will also cause execution of all statements of `AddDigits` itself, since it does result in setting `S1` to **1**. The most important thing to test in `AddDigits`, however, might be the counting within the iteration: is the initial value for `Count`, and the relational operator used for WHILE statement termination correct? Any single test case (if the result is examined, an important part of testing that is sometimes forgotten!) will show this to be correct.

9.3.3 Representation and Comparison of Natural Numbers

The procedures of elementary-school arithmetic operate on natural numbers digit by digit. In CF Pascal, all the digit characters of a number cannot be placed in variables or parameters, because that would limit the maximum size of numbers a procedure could handle. Instead, natural numbers must be stored as strings in TEXT files, and the digits that comprise these strings must be processed a few at a time. Fortunately, the elementary-school processes do confine themselves to one part of the number at a time. Any new natural number, say a sum or product, must likewise be created and written to a file, one digit at a time. There is no need to place any surrounding markers in a file that contains a string representing a natural number in base ten. Leading zeros will be eliminated (except of course if the number is zero), and the automatically added line marker can be detected using EOLN.

The procedure Lexico of Section 9.2.4 compares two strings. It will not properly compare natural numbers represented as strings, however. For example,

lexicographically	†1734† < †268†
numerically	1734 > 268

The trouble is that the digit-by-digit comparisons of lexicographic comparison take no account of the greater weight of leading digits. To compare strings as if they were natural numbers, the following will work:

Compare lengths:
  for unequal lengths, the shorter string is the smaller number;
  for equal lengths, lexicographic comparison is correct.

(This rule fails if care is not taken to suppress extra leading zero digits.) The design is straightforward:

Design Part 5

```
PROCEDURE CompareNaturals(VAR F1, F2: TEXT; VAR Result: CHAR);
  {Compare natural numbers n1, n2 in files F1, F2.
   Set Result to 0,1,2 when n1 is =, <, > n2, respectively.
   F1 and F2 may not be the same file.}
  {Include PROCEDURE Length(VAR F1, F2: TEXT; VAR Result: CHAR)
    Result is 0, 1, 2 if length of F1 =, <, >
    length of F2, respectively.  The actual parameters
    corresponding to F1 and F2 must be distinct.}
  {Include PROCEDURE Lexico(VAR F1, F2: TEXT; VAR Result: CHAR)
    Result is 0, 1, 2 if lexicographic order of
    F1 =, <, > that of F2.  The actual parameters
    corresponding to F1 and F2 must be distinct.}
  BEGIN {CompareNaturals}
    Length(F1, F2, Result);
    IF Result = '0'
    THEN
      Lexico(F1, F2, Result)
  END {CompareNaturals}
```

9.3.4 Addition of Natural Numbers

In adding two natural numbers with pencil and paper we start with the digits at the end of the number strings, not the beginning. As we add each pair of digits, we keep track of whether or not to add an extra 1 (the carry) to the next pair. Thus to add two natural numbers as strings in files, reverse those strings, create the sum string digit by digit keeping track of carries, and finally reverse the sum string. Design Part 6 outlines this strategy, introducing a procedure AddReverse, which will be written to carry out the digit-by-digit add and carry.

Design Part 6

```
PROCEDURE AddNaturals(VAR F1, F2, F3: TEXT);
  {Add natural numbers n1 in F1 and n2 in F2,
   and put the sum n1+n2 in F3.  The actual
   parameters must be distinct.}
  VAR
    T1, T2, T3: TEXT;
  {Include PROCEDURE Reverse(VAR F1, F2: TEXT)
    F2 is characters of string F1 in reverse}
  BEGIN {AddNaturals}
    Reverse(F1, T1);
    Reverse(F2, T2);
    AddReverse(T1, T2, T3);
    Reverse(T3, F3)
  END  {AddNaturals}
```

Procedure `AddReverse` must read digits from from its two input files, then write a sum digit to its output file keeping track of the carry. For example, to add 63725 and 418 the reverse strings are 52736 and 814, which are processed as follows:

Input 1	Input 2	Output	Carry	Explanation
52736	814	3_	1	$5+8=13$
52736	814	34_	0	$2+1+1=4$
52736	814	341_	1	$7+4+0=11$
52736		3414_	0	$3+1=4$
52736		34146_	0	$6+0=6$

In this example Input 2 is exhausted before Input 1, but processing must continue until the end of the longer input is reached. Even if the inputs are of the same length, `AddReverse` must be prepared to add a final carry to the output, as in the addition

$$71 + 65 = 136.$$

Therefore, a strategy for `AddReverse` would be to add and carry so long as neither input is exhausted; then copy in whichever input was not exhausted, being careful with the carry; and lastly insert a final carry, if any. The copying after one input is exhausted is a good candidate for a procedure `AddRemainder`, with a parameter to select the remaining input.

Design Part 6.1

```
PROCEDURE AddReverse(VAR T1, T2, T3: TEXT);
  {Add natural numbers in reverse format in T1, T2 into
   reverse format in T3}
  VAR
    Carry: CHAR;
  BEGIN {AddReverse}
    RESET(T1);
    RESET(T2);
    REWRITE(T3);
    Carry := '0';
    {Place sum digits into T3 so long as both
     T1 and T2 have digits; keep track of carry.};
    AddRemainder(T1, T3, Carry);
    AddRemainder(T2, T3, Carry);
    IF Carry = '1'
    THEN
      WRITE(T3, '1');
    WRITELN(T3)
  END  {AddReverse}
```

Design Part 6.1.1

```
BEGIN {Place sum digits into T3 so long as both
       T1 and T2 have digits; keep track of carry.}
  WHILE NOT EOLN(T1) AND NOT EOLN(T2)
  DO
    BEGIN
      READ(T1,D1);
      READ(T2,D2);
      {D3, Carry := (D1+D2) mod 10, (D1+D2) div 10};
      WRITE(T3,D3)
    END
END
```

Design Part 6.1.1.1

```
BEGIN {D3, Carry := (D1+D2) mod 10, (D1+D2) div 10}
  LastCarry := Carry;
  AddDigits(D1, D2, Carry, D3);
  IF LastCarry = '1'
  THEN
    BEGIN
      IF D3 = '9'
      THEN
        Carry := '1';
      NextDigit(D3)
    END
END
```

Design Part 6.2

```
PROCEDURE AddRemainder(VAR T1, T3: TEXT; VAR Carry: CHAR);
  {Append the remaining digits of T1 to T3
   keeping track of the carry.}
  VAR
    D1, D3: CHAR;
  BEGIN {AddRemainder}
    WHILE NOT EOLN(T1)
    DO
      BEGIN
        READ(T1,D1);
        {D3, Carry := (D1+Carry) mod 10, (D1+Carry) div 10};
        WRITE(T3,D3)
      END;
    READLN(T1)
  END  {AddRemainder}
```

Design Part 6.2.1

```
BEGIN {D3, Carry := (D1+Carry) mod 10, (D1+Carry) div 10}
  IF Carry = '1'
  THEN
    IF D1 = '9'
    THEN
      D3 := '0'
    ELSE
      BEGIN
        Carry := '0';
        NextDigit(D1);
        D3 := D1
      END
  ELSE
    D3 := D1
END
```

This completes the design of `AddNaturals`. In combining the design parts into a CF Pascal text, care must be taken to properly position declarations for variables and procedures introduced in the development.

9.3.5 Multiplication of Natural Numbers

An example of the tabular form of multiplication is:

$$\begin{array}{rr} & 1543 \\ \times & 372 \\ \hline & 3086 \\ & 108010 \\ & 462900 \\ \hline & 573996 \end{array}$$

1543 is the multiplicand and 372 the multiplier, and each digit of the multiplier is used to create a partial product by multiplying it by the multiplicand. These partial products, suitably shifted, are then added to get the final product 573996.

This form of multiplication can be carried out in CF Pascal using only additions. Multiplications by powers of ten can be done by shifting in 0, and multiplications by single digits by repeated addition. In the example, the calculation is as follows:

$$
\begin{aligned}
1543 \times 372 &= 1543 \times 2 \\
&\quad + 1543 \times 70 \\
&\quad + 1543 \times 300 \\
&= (1543 \times 2 \\
&\quad + 15430 \times 7) \\
&\quad + 154300 \times 3 \text{ where} \\
1543 \times 2 &= 1543 + 1543 \\
15430 \times 7 &= 15430 + \ldots + 15430 \text{ (7 terms)} \\
154300 \times 3 &= 154300 + 154300 + 154300
\end{aligned}
$$

The design for multiplication along these lines will omit the file reversals needed to begin and to put the answer in proper form.

Design Part 7

```
PROCEDURE MultiplyReverse(VAR N1, N2, Product: TEXT);
  {Multiply natural numbers in reverse format in N1, N2
   and put the result in reverse format in Product}
  VAR
    D1: CHAR;
    Temp: TEXT;
  BEGIN {MultiplyReverse}
    {Initialize files, put 0 in Product};
    WHILE NOT EOLN(N1)
    DO
      BEGIN
        READ(N1, D1);
        {Product := Product + D1 * N2};
        {N2 := N2 * 10}
      END
  END {MultiplyReverse}
```

Design Part 7.1

```
BEGIN {Initialize files, put 0 in Product}
  RESET(N1);
  RESET(N2);
  REWRITE(Product);
  WRITELN(Product, '0')
END
```

Design Part 7.2

```
BEGIN {Product := Product + D1 * N2};
  WHILE D1 > '0'
  DO
    BEGIN
      AddReverse(N2, Product, Temp);
      RESET(Temp);
      REWRITE(Product);
      CopyOpen(Temp, Product);
      PreviousDigit(D1)
    END
END
```

Design Part 7.3

```
BEGIN {N2 := N2 * 10}
  RESET(N2);
  REWRITE(Temp);
  WRITE(Temp, '0');
  CopyOpen(N2, Temp);
  RESET(Temp);
  REWRITE(N2);
  CopyOpen(Temp, N2)
END
```

Design Part 7.2.1

```
PROCEDURE PreviousDigit(VAR D1: CHAR);
  {((D1 > '0') AND (D1 <= '9') --> D1 := D1 -1) |
   (D1 = 0 --> D1 := '9')}
  BEGIN {PreviousDigit}
    IF D1 = '9' THEN D1 := '8' ELSE
    IF D1 = '8' THEN D1 := '7' ELSE
    IF D1 = '7' THEN D1 := '6' ELSE
    IF D1 = '6' THEN D1 := '5' ELSE
    IF D1 = '5' THEN D1 := '4' ELSE
    IF D1 = '4' THEN D1 := '3' ELSE
    IF D1 = '3' THEN D1 := '2' ELSE
    IF D1 = '2' THEN D1 := '1' ELSE
    IF D1 = '1' THEN D1 := '0' ELSE
    IF D1 = '0' THEN D1 := '9'
  END  {PreviousDigit}
```

9.3.6 A Starter Library for Natural Numbers

The procedures developed in the previous section are the nucleus of a library to operate on natural numbers stored without extra leading zeros as strings in files. The procedure headers are collected below:

```
PROCEDURE NextDigit(VAR D1: CHAR);
  {D1 is replaced by its successor in the
   sequence 0123456789012...}
PROCEDURE AddDigits(VAR D1, D2, S1, S2: CHAR);
  {Add digits D1, D2 to get sum as digits S1, S2
   where S1 is 0 for a one-digit sum in S2, or 1
   for a two-digit sum whose unit digit is S2.}
PROCEDURE CompareNaturals(VAR F1, F2: TEXT; VAR Result: CHAR);
  {Compare natural numbers n1, n2 in files F1, F2.
   Set Result to 0,1,2 when n1 is =, <, > n2, respectively.
   F1 and F2 may not be the same file.}
PROCEDURE AddNaturals(VAR F1, F2, F3: TEXT);
  {Add natural numbers n1 in F1 and n2 in F2,
   and put the sum n1+n2 in F3.  The actual
   parameters must be distinct.}
PROCEDURE AddReverse(VAR T1, T2, T3: TEXT);
  {Add natural numbers in reverse format in T1, T2 into
   reverse format in T3}
```

```
PROCEDURE MultiplyReverse(VAR N1, N2, Product: TEXT);
  {Multiply natural numbers in reverse format in N1, N2
   and put the result in reverse format in Product}
PROCEDURE PreviousDigit(VAR D1: CHAR);
  {((D1 > '0') AND (D1 <= '9') --> D1 := D1 -1) |
   (D1 = 0 --> D1 := '9')}
```

9.3.7 Exercises

9.3.1 Show the sequence of execution states for procedure `AddDigits`, starting with state {D1·**9**, D2·**1**}.

9.3.2 Compute the meaning function for `AddDigits`, by first calculating the meaning of `NextDigit`. Rewrite the summary comment for the procedure making it as complete and formal as possible.

9.3.3 In structurally testing `AddDigits` one might think of separating out the test of `NextDigit`.

a) Write a test-driver program with `NextDigit` in it that will allow a structural test of this procedure to be done in isolation. Specify a complete structural test.

b) Compare the complexity of testing `NextDigit` separately, to testing it within `AddDigits`. Which test do you think is better at finding errors in `NextDigit`?

c) Consider the general case of testing a procedure embedded in a program—will it be harder to generate complete structural tests for it than if it were first isolated? Will the tests be better in isolation? Explain.

9.3.4 Suppose that in `AddDigits`, `S1` were initially set to **1** instead of to **0**. Modify the text so that `AddDigits` still works properly.

9.3.5 Modify procedure `AddDigits` to permit addition of digits in

a) base 2

b) base 8

c) base 16 (with 16 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)

d) Explain why a different approach might be appropriate in a) than in c).

9.3.6 Show the sequence of execution states for each of the following:

a) `CompareNaturals` starting with {F1·<†**2**†,†**3**†,R>, F2·<†**24**†,††,R>}.

b) `AddNaturals` starting with {F1·<†**2**†,†**3**†,**R**>, F2·<†**7**†,††,**R**>, F3·<†**3**†,††,**W**>}.

9.3.7 Calculate the part function for `CompareNaturals`.

9.3.8 Why can't Design Part 6.2.1 be simply:

```
AddDigits(D1, Carry, Carry, D3)
```

and what does this suggest about `AddDigits`?

9.3.9 Show that `AddNaturals` cannot improperly insert a leading zero in its result.

9.3.10 Give a set of structural tests for each of:

a) `CompareNaturals`,

b) `AddNaturals`,

c) `AddReverse`.

9.3.11 When data must be transmitted over uncertain media (for example, radio broadcasts subject to static), it is worth augmenting the data itself with redundant information. One common scheme is to add a "checksum." In the file representation of natural numbers this might be an additional character at the end of each number file, which is the final digit of the sum of all the digits in the number. For example:

Number	*Digit sum*	*Check digit*	*Number with check*
0	0	0	00
123	6	6	1236
57	12	2	572
10091	11	1	100911

a) This scheme isn't foolproof. Give an example of two numbers that differ by just one digit, yet have the same check digit, and explain the significance of such an example.

b) Write a routine to compute the check digit for a given number that lacks it, using procedure `AddDigits`.

c) Write a routine that checks if the last digit of a number is or is not the correct check digit.

d) For two numbers assumed to have check digits, write an add routine, using `AddNatural`, the check routine of c), and as little new code as possible.

9.3.12 Subtraction is usually defined as the partial inverse of addition; that is, x - y is that natural number z such that $z + y = x$, if $x \geq y$; otherwise x - y is not defined in the natural numbers. Use this definition to directly compute x - y, using trial and error with `AddNaturals`. What would your procedure do if it failed to check first for $x \geq y$?

9.3.13 Neither `Length` nor `Lexico` detect the end of a file using `EOLN`, yet this is the convention assumed for string representing natural numbers. Decide whether or not this causes trouble, and explain.

9.3.14 The input files `F1` and `F2` are not `RESET` in `CompareNaturals`. Is this wrong for the assumed conventions? Explain.

9.3.15 The comment on `CompareNaturals` forbids aliasing of the parameters.

a) Explain why this restriction could be troublesome for a user of `CompareNaturals`.

b) Explain why this restriction is needed, giving an example of what would go wrong if it were violated.

c) Repair the procedure so that the restriction is not needed.

9.3.16 Design a procedure to compare natural numbers which examines the natural numbers only once (instead of twice, as `CompareNaturals` sometimes does).

9.3.17 Write a digit×digit multiply routine for the library of Section 9.4.4:

```
PROCEDURE MultiplyDigits(VAR D1, D2, P1, P2: CHAR)
  {Multiply digits D1, D2 to get product P1, P2}
```

and give a more precise comment to describe it.

9.3.18 Write a digit by natural number multiply routine for the library of Section 9.4.4:

```
PROCEDURE MultiplyDigitByNatural(VAR Dig: CHAR; VAR N1, N2: TEXT)
  {Multiply the digit Dig by the contents of file N1,
   result to N2}
```

use any many other library procedures as possible, including the result of the previous exercise.

9.3.19 Use the results of the previous two exercises to rewrite `MultiplyReverse` with less addition.

9.3.20 Utilize `MultiplyReverse` to construct a natural number multiplication routine, as `AddNaturals` was constructed using `AddReverse`.

9.3.21 Using your knowledge of paper-and-pencil subtraction, design a subtraction procedure similar to `AddNaturals` that returns an error character if the numbers are not in the proper order for natural-number subtraction.

9.4 Chapter Summary

Adding parameters to CF Pascal procedures allows a single procedure to be used several times, with an effect appropriate to each application. The meaning of the procedure can be calculated once, and documented so that it need not be checked each time the procedure is used. Aliasing of parameters can cause a difference between the meaning calculated in isolation and that found by repeating the calculation at each application. Aliasing must therefore be documented, and where possible, this unpleasant effect eliminated, if procedures are to be reused.

A procedure has almost exactly the same structure as a program, in that it is a <block> that may contain declarations. The local variables (and local procedures) declared within a procedure are truly only dummy names, that by definition cannot conflict with any others in the program. However, if identifiers are reused, confusing situations can arise in which the scope of an identifier is complicated. This complication is reflected in the meaning of procedures with local variables.

CF Pascal has no built-in operations for manipulating natural numbers, but procedures to compare, add, and multiply numbers of arbitrary size can be written, where the numbers are represented by digit strings in files.

CHAPTER 10

PROGRAMMING WITH WORD STRINGS

Chapter Preview

A library of procedures for text and file manipulation will be constructed. BNF grammars can aid in the statement of a text-manipulation problem and can suggest alternatives to achieve a more effective system design.

10.1 Word Strings

Preview

BNF is an aid to program design when text is involved. A program for removing extra blanks from a string is designed.

10.1.1 Words and Languages

People communicate in spoken and written languages using sequences of sounds and characters. *Words* are the meaningful parts of these sequences. Written words can be found in dictionaries, and words of one language can be translated, more or less accurately, into words with similar meanings in another language.

Words are formed by arranging characters in sequence, but not all such arrangements are words. Similarly, sentences are formed by arranging words in sequence, but not all such arrangements are sentences. One sentence-forming rule in English is that a sentence must have a subject and a predicate. Sentence structure can be shown by BNF syntax rules, as was done for CF Pascal. However, English sentence structure cannot be captured by syntax and context rules, because no complete set of rules exists. English is a natural language which will continue to evolve through human usage. Pascal is a formal language, whose structure is agreed upon and fixed once and for all.

In many languages words are set off from each other by blanks and punctuation marks. In fact, most languages use spaces and punctuation marks that are not available for forming words. Language text in a Pascal file can similarly be analyzed by looking for spaces and punctuation.

Since words have no length limitation, the only way to hold any possible word using CF Pascal is in a file. (If the length of a word is known, a fixed number of CHAR variables can hold the individual characters.) Similarly, files must be used to hold sentences. It is more difficult to process at the word level than character by character, because the individual items are harder to manipulate. But text processing is difficult mostly because many more things can be imagined and usefully carried out at the word level.

10.1.2 Using BNF in Program Design

When text files have a structure that can be described by BNF, the rules of this description will be useful in designing programs to operate on the text. In fact, in word-string problems, it is good practice to work out a BNF description of both the input file and the output file and to refer to that description during program design. Many difficulties that arise in manipulating text result from imperfect understanding of its detailed appearance. A BNF description helps in thinking about the text, because it describes the form exactly.

10.1.3 Removing Extra Blanks

One of the simplest structures a file can have is an alternation of words and blanks, where a word is defined as any string of nonblanks. Perhaps the simplest interesting problem that can be posed for a word/blank file is the removal of extra blanks, so that the words are separated by single blanks, and no blanks precede the first word or follow the last word. This might be done to a file containing a CF Pascal program, for example, to reduce its size.

Consider a text string containing extra blanks in a file `FileIn`:

† **Four score and seven years ago** †

It is to be copied to `FileOut` without the extra blanks:

†**Four score and seven years ago**†

The text in `FileIn` may begin with blanks or with a word; it will always appear to end with a blank resulting from the final line marker. Except for that line mark, the file may be empty. It is remembering all the possibilities that makes solving text problems difficult. BNF can help by capturing all the variations. If we agree not to distinguish between line marks and blanks (since line marks read as blanks), the following rules describe the input:

1. <FileIn> ::= <WBlist> <blank> ¦ <blanks> <WBlist> <blank>
2. <WBlist> ::= <empty string> ¦ <word> <blanks> <WBlist>
3. <word> ::= <nonblank> ¦ <word> <nonblank>
4. <blanks> ::= <blank> ¦ <blanks> <blank>

where <empty string>, <blank>, <nonblank> describe the string of no characters, one blank character, and one nonblank character, respectively. A <WBlist> may be empty, but a <blanks> or a <word> must contain at least one character.

If the output consists of a single line it can be described by just two more syntax rules:

5. <FileOut> ::= <Wlist> ¦ <empty string>
6. <Wlist> ::= <word> ¦ <word> <blank> <Wlist>

That is, a <FileOut> is either empty, or it must begin and end with a <word>, and all adjacent <word>s must be separated by a single blank.

The BNF describes the input and output format precisely, but it does not capture an important part of the problem: it does not state that the words of the output are required to be the same as those of the input, in the same order. This kind of statement must be made with a context rule:

CR. The <word>s of <Wlist> must be the same as those of <WBlist>, in the same order.

In thinking how to read `FileIn`, remove the extra blanks, and write `FileOut`, two useful program patterns are flushing blanks, and copying a word. Using a `CHAR` variable `Ch` with initial value □, the program part:

```
{flush blanks}
WHILE NOT EOF(FileIn) AND (Ch = ' ')
DO
  READ(FileIn,Ch);
```

will begin anywhere in `FileIn`, and advance to the first nonblank character, which will be held in `Ch` when the WHILE statement terminates. If it happens that the next character in `FileIn` is nonblank, that is, there are no blanks to flush, that character is read into `Ch` and the WHILE statement terminates.

When `Ch` contains the initial character of a word (as it will if blanks have just been flushed), that word can be copied to `FileOut` by:

```
{copy word}
WHILE Ch <> ' '
  DO
    BEGIN
      WRITE(FileOut,Ch);
      READ(FileIn,Ch)
    END
```

When Ch has the value □, the WHILE statement is never executed, so nothing is done, the right action for copying a nonexistent word. Since a TEXT file ends with a line marker (which appears as a blank), the end of file cannot be encountered while copying a word.

The program parts that flush blanks and copy a word can be used as high-level operations in processing files, in particular for designing a procedure RemoveExtraBlanks. According to BNF rule 1 describing the input, there are two possible cases a design must face in a <FileIn>, namely:

1. <WBlist> <blank>
2. <blanks> <WBlist> <blank>

Case 2 is reduced to case 1 by initially flushing blanks. Whether or not there are blanks, the first character of <WBlist> will be read into Ch. Thus the initial design is:

Design Part 1

```
PROCEDURE RemoveExtraBlanks(VAR FileIn, FileOut: TEXT);
  {Copy FileIn words to FileOut, removing leading and
   trailing blanks, and extra blanks between words}
  VAR
    Ch: CHAR;
  PROCEDURE FlushBlanks(VAR Ch: CHAR);
    BEGIN {FlushBlanks}
      WHILE NOT EOF(FileIn) AND (Ch = ' ')
      DO
        READ(FileIn,Ch)
    END;  {FlushBlanks}
  BEGIN {RemoveExtraBlanks}
    {initialize};
    FlushBlanks(Ch);
    {remove extra blanks for case 1};
    WRITELN(FileOut)
  END  {RemoveExtraBlanks}
```

where the program part for flushing blanks has been made into a procedure.

Design Part 1.1

```
BEGIN {initialize}
  RESET(FileIn);
  REWRITE(FileOut);
  Ch := ' '
END
```

To remove extra blanks for case 1, the <WBlist> term is expanded using rule 2, resulting in two cases:

1.1 <blank>
1.2 <word> <blanks> <WBlist> <blank>

The sequence:

```
{copy word};
FlushBlanks(Ch)
```

will reach either a <WBlist> or the end of file in every case. If a <WBlist> is reached (case 1.2), this sequence should be repeated; if the end of file is reached (case 1.1), processing should terminate. Thus these operations should be placed in a WHILE statement with the termination condition EOF(FileIn).

After each word but the last is copied, a single blank must be written to separate the words, according to BNF rules 5 and 6 describing the output. The last word can be identified by testing for end of file following FlushBlanks, which will skip the line marker at the end. This analysis is summarized in the following design part.

Design Part 1.2

```
BEGIN {remove extra blanks for case 1}
  WHILE NOT EOF(FileIn)
  DO
    BEGIN
      {copy word};
      FlushBlanks(Ch);
      {insert blank if not last word}
      IF NOT EOF(FileIn)
      THEN
        WRITE(FileOut,' ')
    END
END
```

The complete design for RemoveExtraBlanks is:

```
PROCEDURE RemoveExtraBlanks(VAR FileIn, FileOut: TEXT);
  {Copy FileIn words to FileOut, removing leading and
   trailing blanks, and extra blanks between words}
  VAR
    Ch: CHAR;
  PROCEDURE FlushBlanks(VAR Ch: CHAR);
    BEGIN {FlushBlanks}
      WHILE NOT EOF(FileIn) AND (Ch = ' ')
      DO
        READ(FileIn,Ch)
    END;  {FlushBlanks}
  BEGIN {RemoveExtraBlanks}
    BEGIN {initialize}
      RESET(FileIn);
      REWRITE(FileOut);
      Ch := ' '
    END;
    FlushBlanks(Ch);
    BEGIN {remove extra blanks for case 1}
      WHILE NOT EOF(FileIn)
      DO
        BEGIN
          BEGIN {copy word}
            WHILE Ch <> ' '
              DO
                BEGIN
                  WRITE(FileOut,Ch);
                  READ(FileIn,Ch)
                END
```

```
            END;
            FlushBlanks(Ch);
            {insert blank if not last word}
            IF NOT EOF(FileIn)
            THEN
              WRITE(FileOut,' ')
          END
      END;
      WRITELN(FileOut)
    END  {RemoveExtraBlanks}
```

10.1.4 Testing RemoveExtraBlanks

In order to test `RemoveExtraBlanks`, it would be useful to mark the beginning and end of strings so that blanks are easy to spot. The character # will serve this purpose. By reading the test string from INPUT and writing the results to OUTPUT, many cases can be tried at once, as in this test program:

```
PROGRAM TestREB(INPUT,OUTPUT);
  VAR Ch: CHAR;
    TestIn, TestOut: TEXT;
  {Include
   PROCEDURE RemoveExtraBlanks(VAR FileIn, FileOut: TEXT);
     Copy FileIn words to FileOut, removing leading and
     trailing blanks, and extra blanks between words}
  BEGIN {TestREB}
    WHILE NOT EOF
    DO
      BEGIN
        {Copy one line of INPUT to a file, with echo}
        REWRITE(TestIn);
        WRITE('Input is #');
        WHILE NOT EOLN
        DO
          BEGIN
            READ(Ch);
            WRITE(Ch);
            WRITE(TestIn, Ch);
          END;
        READLN;
        WRITELN('#');
        WRITELN(TestIn);
        RemoveExtraBlanks(TestIn, TestOut);
        {Copy result to OUTPUT}
        RESET(TestOut);
        WRITE('Output is #');
        WHILE NOT EOLN(TestOut)
        DO
          BEGIN
            READ(TestOut, Ch);
            WRITE(Ch)
          END;
        WRITELN('#')
      END
  END. {TestREB}
```

Execution

```
INPUT :  Pack my   box  with six.
         SOLIDPACKED□□
OUTPUT:Input is #  Pack my   box  with six.#
       Output is #Pack my box with six.#
       Input is #SOLIDPACKED  #
       Output is #SOLIDPACKED#
```

The program passes each input line to RemoveExtraBlanks and prints the result. The BNF description of all possible inputs (rules 1 and 2) provides a way to select test data. If we believe that the parts of RemoveExtraBlanks for flushing blanks and copying words work properly, what must be tested is the combinations of <word> and <blanks> positions described by those rules. There are four patterns possible from the rules:

<blank>
<word> <blanks> <WBlist> <blank>
<blanks> <blank>
<blanks> <word> <blanks> <WBlist> <blank>

Particular test cases fitting each pattern, except for a final <WBlist> <blank>, are:

††
†**the** †
† †
† **the** †

In addition, a string fitting the second pattern, in which that pattern is repeated several times, is

†**The quick brown fox jumped** †

Thus if the final <WBlist> is chosen as <empty string> and the final <blank> is supplied by the line mark that terminates the input, these cases try the four possible BNF descriptions.

Testing in this way is called *partition testing,* because the input space is divided up into groups of strings that are expected to be handled in the same way. Partition testing is very different from structural testing. First, it relies not on the program, but on the problem definition. It can therefore be planned before a program is ever written. Second, it has a chance of detecting *omissions* in the program (as structural testing cannot hope to do). The cases selected represent all the input possibilities, so if the program fails to handle one, the error will be caught. The method is not perfect, however, because it does not try *all* values for <WBlist> in each possible position. It could happen that the program fails only (say) when there are more than 1000 words in the input. But if the program was designed systematically, such failures are unlikely, and partition testing is a good method of finding errors.

10.1.5 Analyzing RemoveExtraBlanks

To demonstrate that `RemoveExtraBlanks` does what it was designed to do, its part function can be calculated. The design for the DO part of the inner WHILE statement is:

```
{copy word};
FlushBlanks(Ch);
{insert blank if not last word}
```

The intended function for each of these three parts can be written as a conditional assignment.

The intended part function C for copying a word is:

$$C = C1 \mid C2$$

where

$$C1 = ((Ch <> \text{' '}) \text{ AND } (\text{NOT}(\text{EOF}(\text{FileIn}))) \text{ AND } (\text{FileIn.2} = (x \bigtriangledown B) \ \& \ y) \text{ AND } (\text{FileIn.3} = \mathbf{R}) \text{ AND } (\text{FileOut.3} = \mathbf{W}) \rightarrow \text{Ch,FileOut.1,FileIn.1,FileIn.2} := \square, (\text{FileOut.1} \bigtriangledown \text{Ch}) \ \& \ x, \text{FileIn.1} \ \& \ (x \bigtriangledown B), y)$$

and

$$C2 = (\text{Ch} = \text{' '} \rightarrow \ )$$

where x is a string of nonblank characters containing no line marker, y is a string, and B is either a line marker or $\square$. In this conditional assignment and the ones to follow, the members of the 3-list file values are indicated by the subscript following a dot. For example, `FileIn.2` means FileIn_2, the second member (which is the future string) of this file value. This notation allows conditional assignments to be written as comments in programs, where subscripts are not available.

The intended function F for `FlushBlanks(Ch)` is:

```
F = (F0 → (F1 | F2)) | F3
```

where

```
F0 = ((Ch = ' ') AND (NOT(EOF(FileIn))) AND (FileIn.3 = R)
F1 = (FileIn.2 = b ∇ Z & w →
     Ch, FileIn.1, FileIn.2 := Z, FileIn.1 & b ∇ Z, w)
F2 = (FileIn.2 = b ∇ / →
     Ch, FileIn.1, FileIn.2 := □, FileIn.1 & FileIn.2, ††))
F3 = ((Ch <> ' ') OR (EOF(FileIn)) →   )
```

where b is a string of blank and line-marker characters (possibly empty), Z is a character other than blank or a line marker, and w is a nonempty string of characters.

The proofs that

$$\boxed{\texttt{BEGIN \{copy word\} ... END}} = C$$

and

$$\boxed{\texttt{FlushBlanks(Ch)}} = F$$

are similar to the one in Section 8.2.3 for a fragment of code that copies nonblank characters, so they are omitted.

The proof that I below is the meaning for the conditional blank insertion is straightforward.

```
I = I1 | I2
```

where

```
I1 =
  ((NOT(EOF(FileIn))) AND (FileOut.3 = W) AND (FileIn.3 = R →
      FileOut.1 := FileOut.1 ∇ □)
I2 = ((EOF(FileIn)) AND (FileIn.3 = R) →   )
```

The meaning of the DO part is therefore the composition of these three functions $C \circ F \circ I$. The composition can be calculated using conditional trace tables. Each condition is labeled; there are two for C, three for F, and two for I, for 12 total combinations. Each table will be labeled with the combination it covers. For example, $(C1,F1,I1)$ represents the first condition of C, the first condition of F, and the first condition of I. `FileIn` and `FileOut` will be abbreviated by `FI` and `FO`, respectively.

The 12 cases are:

Case 1 $(C1,F1,I1)$	Case 5 $(C1,F3,I1)$	Case 9 $(C2,F2,I1)$
Case 2 $(C1,F1,I2)$	Case 6 $(C1,F3,I2)$	Case 10 $(C2,F2,I2)$
Case 3 $(C1,F2,I1)$	Case 7 $(C2,F1,I1)$	Case 11 $(C2,F3,I1)$
Case 4 $(C1,F2,I2)$	Case 8 $(C2,F1,I2)$	Case 12 $(C2,F3,I2)$

Case 1 (*C1,F1,I1*)

Condition	Ch	FO.1	FI.1	FI.2
(Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.2 = $x \nabla B$ & y) AND (FI.3 = **R**) AND (FO.3 = **W**)	□	FO.1 ∇ Ch & x	FI.1 & $x \nabla B$	y
(□ = ' ') AND (NOT(EOF(<FI.1 & $x \nabla B, y$, FI.3>))) AND (FI.3 = **R**) AND ($y = b \nabla Z$ & w)	Z		FI.1 & $x \nabla B$ & $b \nabla Z$	w
NOT(EOF(<FI.1 & $x \nabla B$ & $b \nabla Z$, w, FI.3>)) AND (FI.3 = **R**) AND (FO.3 = **W**)		FO.1 ∇ Ch & $x \nabla$ □		

Condition: (Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.2 = $x \nabla B$ & y) AND (FI.3 = **R**) AND (FO.3 = **W**) AND (□ = □) AND (NOT(EOF(<FI.1 & $x \nabla B$, y, FI.3>))) AND ($y = b \nabla Z$ & w) AND (NOT(EOF(<FI.1 & $x \nabla B$ & $b \nabla Z$, w, FI.3>)))

Several parts of this condition are equivalent to TRUE.

Condition	TRUE because
□ = □	Obvious.
NOT(EOF(<FI.1 & $x \nabla B$, y, FI.3>))	By the next condition, $y = b \nabla Z$ & w and w is not empty.
NOT(EOF(<FI.1 & $x \nabla B$ & $b \nabla Z$, w, FI.3>))	w is nonempty.

Thus, substituting for y, the condition reduces to:

Condition: (Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.2 = $x \nabla B$ & $b \nabla Z$ & w) AND (FI.3 = **R**) AND (FO.3 = **W**)

Assignment: Ch, FO.1, FI.1, FI.2 :=
Z, FO.1 ∇ Ch & $x \nabla$ □, FI.1 & $x \nabla B$ & $b \nabla Z$, w

Case 2 (*C1,F1,I2*)

Condition	Ch	FO.1	FI.1	FI.2
(Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.2 = $x \nabla B$ & y) AND (FI.3 = **R**) AND (FO.3 = **W**)	□	FO.1 ∇ Ch & x	FI.1 & $x \nabla B$	y
(□ = ' ') AND (NOT(EOF(<FI.1 & $x \nabla B, y$, FI.3>))) AND (FI.3 = **R**) AND ($y = b \nabla Z$ & w)	Z		FI.1 & $x \nabla B$ & $b \nabla Z$	w
(EOF(<FI.1 & $x \nabla B$ & $b \nabla Z$, w, FI.3>)) AND (FI.3 = **R**)				

Since w contains at least one character, EOF(<FI.1 & $x \nabla B$ & $b \nabla Z$, w, FI.3>) has the value FALSE. Thus Case 2 of the composition does not occur.

Case 3 (*C1,F2,I1*)

Condition	Ch	FO.1	FI.1	FI.2
(Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.2 = $x \nabla B$ & y) AND (FI.3 = **R**) AND (FO.3 = **W**)	□	FO.1 ∇ Ch & x	FI.1 & $x \nabla B$	y
(□ = ' ') AND (NOT(EOF(<FI.1 & $x \nabla B, y$, FI.3>))) AND (FI.3 = **R**) AND ($y = b \nabla$ /)	□		FI.1 & $x \nabla B$ & y	††
NOT(EOF(<FI.1 & $x \nabla B$ & y, ††, FI.3>)) AND (FI.3 = **R**) AND (FO.3 = **W**)		FO.1 ∇ Ch & $x \nabla$ □		

In the *F2* portion of the function *F*, FileIn.2 acquires the value ††, so EOF(FileIn) has value TRUE. Thus the condition of *I1* for conditionally inserting blanks cannot hold, and Case 3 does not occur.

Case 4 (*C1,F2,I2*)

Condition	Ch	FO.1	FI.1	FI.2
(Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.2 = $x \nabla B$ & y) AND (FI.3 = **R**) AND (FO.3 = **W**)	□	FO.1 ∇ Ch & x	FI.1 & $x \nabla B$	y
(□ = ' ') AND (NOT(EOF(<FI.1 & $x \nabla B, y$, FI.3>))) AND (FI.3 = **R**) AND ($y = b \nabla$ /)	□		FI.1 & $x \nabla B$ & y	††
(EOF(<FI.1 & $x \nabla B$ & y, ††, FI.3>)) AND (FI.3 = **R**)				

Condition: (Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.2 = $x \nabla B$ & y)
AND (FI.3 = **R**) AND (FO.3 = **W**) AND (□ = □)
AND (NOT(EOF(<FI.1 & $x \nabla B$, y, FI.3>)))
AND ($y = b \nabla$ /) AND (FI.3 = **R**)
AND (EOF(<FI.1 & $x \nabla B$ & y, ††, FI.3>)) AND (FI.3 = **R**)

which reduces to:

Condition: (Ch <> ' ') AND (NOT(EOF(FI))) AND
(FI.2 = $x \nabla B$ & $b \nabla$ /) AND (FI.3 = **R**) AND (FO.3 = **W**)

by arguments similar to those for Case 1.

Assignment: Ch,FO.1,FI.1,FI.2 :=
□,FO.1 ∇ Ch & x,FI.1 & $x \nabla B$ & $b \nabla$ /,††

Case 5 (*C1,F3,I1*)3

Condition	Ch	FO.1	FI.1	FI.2
(Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.2 = $x \nabla B$ & y) AND (FI.3 = **R**) AND (FO.3 = **W**)	□	FO.1 ∇ Ch & x	FI.1 & $x \nabla B$	y
(□ <> □) OR (EOF(<FI.1 & $x \nabla B$, y, FI.3>))				
NOT(EOF(<FI.1 & $x \nabla B$, y, FI.3>)) AND (FI.3 = **R**) AND (FO.3 = **W**)				

Not all the state information has been filled in, because the condition for *F3* requires that either Ch <> ' ' or EOF(<FI.1 & $x \nabla B$, y, FI.3>) have value TRUE. Under the conditions established by *C1*, the value of Ch is □. Hence the EOF condition must be the one that holds.

However, *I1* requires its negation. Thus Case 5 cannot occur.

Case 6 (*C1,F3,I2*)

Condition	Ch	FO.1	FI.1	FI.2
(Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.2 = $x \nabla B$ & y) AND (FI.3 = **R**) AND (FO.3 = **W**) (□ <> □) OR (EOF(<FI.1 & $x \nabla B$, y, FI.3>)) EOF(<FI.1 & $x \nabla B$, y, FI.3>) AND (FI.3 = **R**)	□	FO.1 ∇ Ch & x	FI.1 & $x \nabla B$	y

C1 leaves Ch with the value □. In *F3*, for EOF(<FI.1 & $x \nabla B, y,$ FI.3>) to have value TRUE, y must be the empty string and B must be a line marker.

Condition: (Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.3 = **R**) AND (FO.3 = **W**) AND (FI.2 = $x \nabla B$ & y) AND (y = ††)

which reduces to:

Condition: (Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.3 = **R**) AND (FO.3 = **W**) AND (FI.2 = $x \nabla$ /)

Assignment: Ch,FO.1,FI.1,FI.2 := □,FO.1 ∇ Ch & x,FI.1 & $x \nabla$ /,††

Case 7 (*C2,F1,I1*)

Condition	Ch	FO.1	FI.1	FI.2
Ch = ' '				
(Ch = ' ') AND (NOT(EOF(FI))) AND (FI.3 = **R**) AND (FI.2 = $b \nabla Z$ & w)	Z		FI.1 & $b \nabla Z$	w
NOT(EOF(<FI.1 & $b \nabla Z$, w, FI.3>)) AND (FI.3 = **R**) AND (FO.3 = **W**)		FO.1 ∇ □		

Condition: (Ch = ' ') AND (NOT(EOF(FI))) AND (FI.3 = **R**) AND (FI.2 = $b \nabla Z$ & w) AND (NOT(EOF(<FI.1 & ∇Z, w, **R**>))) AND (FO.3 = **W**) AND (FI.3 = **R**)

which reduces to:

Condition: (Ch = ' ') AND (NOT(EOF(FI))) AND (FI.3 = **R**) AND (FI.2 = $b \nabla Z$ & w) AND (FO.3 = **W**)

Assignment: Ch,FO.1,FI.1,FI.2 := Z,FO.1 ∇ □,FI.1 & b ∇ Z, w

In **Case 8** (*C2,F1,I2*) as in Case 2, *F1* leaves w (a nonempty string) in FileIn.2 but *I2* requires that FileIn.2 be empty. Thus Case 8 cannot occur.

In **Case 9** (*C2,F2,I1*) as in Case 3, *F2* leaves an empty string in FileIn.2. Thus the NOT(EOF(...)) part of the guard for *I1* cannot be satisfied, and Case 9 cannot occur.

Case 10 (*C2,F2,I2*)

Condition	Ch	FO.1	FI.1	FI.2
Ch = ' '				
(Ch = ' ') AND (NOT(EOF(FI))) AND (FI.3 = **R**) AND (FI.2 = b ∇ /)	□		FI.1 & FI.2	††
(EOF(<FI.1 & FI.2, ††, FI.3>)) AND (FI.3 = **R**)				

Condition: (Ch = ' ') AND (NOT(EOF(FI))) AND (FI.3 = **R**) AND (FI.2 = b ∇ /) AND (EOF(<FI.1 & FI.2, ††, **R**>)) AND (FI.3 = **R**)

which reduces to:

Condition: (Ch = ' ') AND (NOT(EOF(FI))) AND (FI.3 = **R**) AND (FI.2 = b ∇ /)

Assignment: Ch,FI.1,FI.2 := □,FI.1 & FI.2,††

Case 11 (*C2,F2,I1*)

Condition	Ch	FO.1	FI.1	FI.2
Ch = ' '				
(Ch = ' ') OR (EOF(FI))				
(NOT(EOF(FI))) AND (FI.3 = **R**) AND (FO.3 = **W**)				

The condition is that

(Ch = ' ') AND ((Ch <> ' ') OR (EOF(FI))) AND ((NOT(EOF(FI))) AND ...)

which has the value FALSE. Thus Case 11 does not occur.

Case 12 (*C2,F2,I2*)

Condition	Ch	FO.1	FI.1	FI.2
Ch = ' '				
(Ch = ' ') OR (EOF(FI))				
(EOF(FI)) AND (FI.3 = **R**)				

Condition: (Ch = ' ') AND (EOF(FI)) AND (FI.3 = **R**)

Assignment:

Collecting together the cases that can occur, the composition of the three functions $C \circ F \circ$ has the conditions and assignments:

(Case 1)

Condition: (Ch <> ' ') AND (NOT(EOF(FI))) AND
(FI.2 = $x \nabla B \,\&\, b \nabla Z \,\&\, w$) AND (FI.3 = **R**) AND (FO.3 = **W**)

Assignment: Ch, FO.1, FI.1, FI.2 :=
Z,FO.1 ∇ Ch & $x \nabla \square$, FI.1 & $x \nabla B \,\&\, b \nabla Z, w$

(Case 4)

Condition: (Ch <> ' ') AND (NOT(EOF(FI))) AND
(FI.2 = $x \nabla B \,\&\, b \nabla$ /) AND (FI.3 = **R**) AND (FO.3 = **W**)

Assignment: Ch,FO.1,FI.1,FI.2 :=
$\square$,FO.1 ∇ Ch & x,FI.1 & $x \nabla B \,\&\, b \nabla$ /,††

(Case 6)

Condition: (Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.3 = **R**)
AND (FO.3 = **W**) AND (FI.2 = $x \nabla$ /)

Assignment: Ch,FO.1,FI.1,FI.2 := $\square$,FO.1 ∇ Ch & x,FI.1 & $x \nabla$ /,††

(Case 7)

Condition: (Ch = ' ') AND (NOT(EOF(FI))) AND (FI.3 = **R**)
AND (FI.2 = $b \nabla Z \,\&\, w$) AND (FO.3 = **W**)

Assignment: Ch,FO.1,FI.1,FI.2 := Z,FO.1 $\nabla \square$,FI.1 & $b \nabla Z, w$

(Case 10)

Condition: (Ch = ' ') AND (NOT(EOF(FI))) AND (FI.3 = **R**)
AND (FI.2 = $b \nabla$ /)

Assignment: Ch,FI.1,FI.2 := $\square$,FI.1 & FI.2,††

(Case 12)

Condition: (Ch = ' ') AND (EOF(FI)) AND (FI.3 = **R**)

Assignment:

Ignoring the conditions that the input file be readable and the output writeable, cases 1, 4, and 6 apply when the value of Ch is not $\square$, and the input not at end of file, according to the value of FileIn.2:

Case	FileIn.2	Following word x	Added to FileOut.1
1	$x \nabla B \,\&\, b \nabla Z \,\&\, w$	<blank>-<word> pairs	∇ Ch & $x \nabla \square$
4	$x \nabla B \,\&\, b \nabla /$	<blanks>	∇ Ch & x
6	$x \nabla /$	final line marker	∇ Ch & x

x is a string of nonblank characters containing no line marker; B is a blank or line marker; b is a string of blanks or line markers; w is a nonempty string; Z is a character that is neither blank or a line marker.

Cases 7 and 10 apply when the value of Ch is $\square$, and the input is not at end of file:

Case	FileIn.2	Following <blanks> b	Added to FileOut.1
7	$b \nabla Z \,\&\, w$	<word>-<blank> pairs	$\nabla \square$
10	$b \nabla /$	final line marker	

Case 12 applies when the input is at end of file; nothing is added to FileOut.

The part function $g = C \circ F \circ I$ can be written as a conditional assignment by combining similar cases above:

```
g =
((Ch <> ' ') AND (NOT(EOF(FI))) AND (FI.3 = R) AND (FO.3 = W) →
    (1) (FI.2 = x ∇B & b ∇Z & w →
        Ch,FO.1,FI.1,FI.2 := Z,FO.1 ∇ Ch & x ∇ □,
            FI.1 & x ∇B & b ∇Z, w) |
    (4) (FI.2 = x ∇B & b ∇ / →
        Ch,FO.1,FI.1,FI.2 := □,FO.1 ∇ Ch & x,
            FI.1 & x ∇B & b ∇ /,††) |
    (6) (FI.2 = x ∇ / →
        Ch,FO.1,FI.1,FI.2 := □,FO.1 ∇ Ch & x,FI.1 & x ∇ /, ††)) |
((Ch = ' ') AND (NOT(EOF(FI))) AND (FI.3 = R) →
    (7) ((FI.2 = b ∇Z & w) AND (FO.3 = W) →
        Ch,FO.1,FI.1,FI.2 := Z,FO.1 ∇ □,FI.1 & b ∇Z, w) |
    (10) (FI.2 = b ∇ / → Ch,FI.1,FI.2 := □,FI.1 & FI.2,††)) |
((Ch = ' ') AND (EOF(FI)) AND (FI.3 = R) → )
    (12)
```

The numbers at the left are to identify the cases for the discussion below. This part function describes copying a word from FileIn to FileOut, starting with the character value of Ch (unless it is blank); when Ch is blank, a single blank is added to FileOut unless FileIn is positioned at the end of file.

In the design of RemoveExtraBlanks, the input and output files were described using BNF notation. Strings described by BNF can also be described using the formal string operations, as in the definition of g. The correspondence between these two descriptions helps in understanding why the program solves the problem. In Case 1, the guard shows that the input string is:

∇ Ch & $x \nabla B \,\&\, b \nabla Z \,\&\, w$.

Up to the final string w this can be described in BNF notation as:

<word> <blanks> <nonblank>

where <word> is a nonempty string of nonblank characters, <blanks> is a nonempty string of $\square$, and <nonblank> is a nonblank character. (B and b may have line markers, but these will be converted to blanks before they can be examined in Ch.)

The output string for Case 1 of g is modified by adding:

∇ Ch & x ∇ □,

which can be described in the BNF as:

<word> <blank>

In Case 1, g converts a string described by

<word> <blanks> <nonblank>

stored in Ch and part of FileIn into a string of the form

<word> <blank> <nonblank>

in part of FileOut and Ch.

Similarly, in Case 4 of g strings of the form

<word> <blanks>

in Ch and FileIn are converted into strings of the form

<word>

in FileOut. Case 6 converts strings of the form <word> <blank> to the form <word>.

Cases 7 and 10 occur when Ch contains a blank character. Case 7 squeezes multiple blanks to a single blank, converting strings of the form

<blanks> <nonblank>

into strings of the form <blank>. Case 10 removes trailing blanks, converting strings of the form <blanks> to <empty string>.

Case 12 takes no action.

The table below summarizes these conversions from strings in Ch and part of FileIn to strings in part of FileOut.

BNF description of the DO part function g

Case	Ch, FileIn	FileOut	Ch	EOF
1	<word> <blanks> <nonblank>	<word> <blank>	<nonblank>	FALSE
4	<word> <blanks>	<word>	<blank>	TRUE
6	<word> <blank>	<word>	<blank>	TRUE
7	<blanks> <nonblank>	<blank>	<nonblank>	FALSE
10	<blanks>	<empty string>	<blank>	TRUE
12	<blank>	<empty string>	<blank>	TRUE

The DO part of the main WHILE statement in RemoveExtraBlanks has part function g derived above. The BNF table describing g helps in understanding the iteration. The WHILE statement is:

```
WHILE NOT EOF(FileIn)
DO
  {Compute g}
```

Any of the six patterns may occur in the input, and g will produce the corresponding pattern in the output, setting up the next pattern, until end of file is reached on the input.

Using the table, a BNF description of the files actually processed by the WHILE statement can be obtained. Wherever <nonblank> appears in Ch and part of FileIn, the pattern can repeat. Let <R> stand for the strings involved. Only cases 1 and 4 are involved, because only they begin with a <word> that the <nonblank> begins. (Case 6 is a less general version of case 4.) Thus the recursive rule is:

<R> ::= <word> <blanks> <R> (1)
¦ <word> <blanks> (4)

Cases 7 and 10 can apply only initially, since they require Ch to be blank, and no case leads to this situation—all the cases that produce a blank in Ch occur at end of file, where the WHILE statement terminates. Thus if <File1> is the input to the WHILE statement:

<File1> ::= <word> <blanks> <R> (1)
¦ <word> <blanks> (4)
¦ <blanks> <R> (7)
¦ <blanks> (10)

(Case 12 is a less general version of case 10.) It can be seen that the first two alternatives are just <R>, so

<File1> ::= <R> (1,4)
¦ <blanks> <R> (7)
¦ <blanks> (10)

The output file created by the program can be described using BNF for <File2> in a similar way. Cases 4 and 6 are duplicates, as are 10 and 12. The repetition occurs in cases 1 and 4 as before, leading to strings <S>:

<S> ::= <word> <blank> <S> (1)
¦ <word> (4)

The other cases can appear only initially:

<File2> ::= <word> <blank> <S> (1)
¦ <word> (4)
¦ <blank> <S> (7)
¦ <empty string> (10)

Or inserting <S> for the first two alternatives:

<File2> ::= <S> (1,4)
¦ <blank> <S> (7)
¦ <empty string> (10)

It is evident from the table for *g* that the <word>s from <File1> are moved to <File2> in the order they appear. Thus the function *f* of the WHILE loop converts <File1> to <File2> preserving the <word> order if EOF(FileIn) is FALSE, and is the identity otherwise, i.e.

```
f = ((NOT(EOF(FileIn)) AND FileIn.3=R AND FileOut.3=W) →
        FileIn,FileOut := <FileIn.1 & FileIn.2,††,R>,
                          <FileOut.1 & convert(FileIn.2),††,W>) |
    (EOF(FileIn) AND FileIn.3=R →    )
```

where *convert* takes as an argument a string of the form <File1> and produces as value a string of the form <File2>, subject to the condition that their <word>s are the same, in the same order.

A formal proof that *f* is the function performed by the WHILE statement will not be given, but the three conditions of the WHILE verification rule will be discussed informally.

1. domain(*f*) = domain(WHILE...)

From the definition, domain(*f*) is those states for which

```
(FileIn.3=R AND FileOut.3=W AND NOT (EOF(FileIn)))
    OR (FileIn.3=R AND EOF(FileIn))
```

is TRUE. The WHILE statement terminates immediately if EOF(FileIn). For this condition to be defined, FileIn must be open for reading. If EOF(FileIn) is FALSE, at least one character is read (by Flushblanks if a blank, otherwise by {copy word}), so FileIn must be the in the

R mode. Similarly, the WRITE statements of {copy word} require FileOut to be in the **W** mode. The WHILE statement cannot read past end of file. FlushBlanks checks for end of file, and {copy word} stops on a blank, which must end the file because of the final line marker. Thus domain(WHILE ...) is the same as domain(f).

2. (EOF(FileIn) $\rightarrow f$) = (EOF(FileIn) $\rightarrow$)

In states where EOF(FileIn) has value TRUE, the second condition of f applies, where f is the identity as required.

3. f = IF NOT EOF(FileIn) THEN {compute g} $\circ f$

The table above shows that g transfers a single <word>, thus since f transfers sequences of <word>s, the same result is obtained by applying f, or by first applying g and then f.

In terms of the WHILE-statement function f, RemoveExtraBlanks has the form:

```
BEGIN {RemoveExtraBlanks}
  BEGIN {initialize}
    RESET(FileIn);
    REWRITE(FileOut);
    Ch := ' '
  END;
  FlushBlanks(Ch);
  {compute f};
  WRITELN(FileOut)
END  {RemoveExtraBlanks}
```

The initialization statements establish a state s with the partial contents

{FileIn·<††,x,**R**>, FileOut·<††,††,**W**>, Ch·□},

where x is the input string. FlushBlanks has a meaning with three cases *F1*, *F2*, *F3* given above. The conditional trace tables for FlushBlanks(Ch) beginning in state s are:

Case *F1*

Condition	Ch	FI.1	FI.2
	□	††	$b \nabla Z$ & w
(Ch = ' ') AND (NOT EOF(FI))	Z	FI.1 & $b \nabla Z$	w

Thus in Case *F1*, Ch becomes the first nonblank character, and the remainder of input w is nonempty.

Case *F2*

Condition	Ch	FI.1	FI.2
	□	††	$b \nabla /$
(Ch = ' ') AND (NOT EOF(FI))	□	$b \nabla /$	††

Case *F2* adds nothing to the output, because the entire input string of blanks is flushed.

Case *F3*

Condition	Ch	FI.1	FI.2
	□	††	††
(Ch <> ' ') OR (EOF(FI))			

Thus after the initialization statements and FlushBlanks(Ch) have been executed, the state either has EOF(FileIn) TRUE, or Ch has as value the first nonblank character in FileIn. Thus the function f for the WHILE statement is not applied to any leading blanks in FileIn, and case 7 of the DO part function g (where a single blank was transferred) cannot arise. The description of the program's action in BNF can now be completed. Eliminating the case 7 rule from

the WHILE statement description,

<File1> ::= <R> ¦ <blanks>
<R> ::= <word> <blanks> <R> ¦ <word> <blanks>
<File2> ::= <S> ¦ <empty string>
<S> ::= <word> <blank> <S> ¦ <word>

The initial `Flushblanks` adds a case to the input, but not the output:

<FileIn> ::= <File1> ¦ <blanks> <File1>
<FileOut> ::= <File2>

It can be seen that this describes the same file structure as the original specification.

This concludes the analysis of `RemoveExtraBlanks`. The treatment has not been completely formal, but the program calculus has been used to gain a solid understanding of each program part and of their effect in combination. In analyzing a program, analysis can use more or less rigor. The goal is to understand, with appropriate confidence, the effect of the program. Sometimes the analysis requires a great deal of rigor and sometimes informal analysis is sufficient. For example, if the program part is difficult to understand or life-critical (for example, the control program for a manned space flight), then a detailed analysis might be required to assure confidence in the correctness of the program. On the other hand if the program part is readable and easy to understand, then going through a rigorous analysis would not provide a proportionately greater level of confidence for the cost involved. In practice, the level of rigor must be weighed against the cost.

Program analysis makes use of many different formalisms. This example used string notation and BNF notation as well as conditional assignments and trace tables. Formalisms are invented to make the task of difficult analysis easier, and new ones may be invented for other problems.

10.1.6 Exercises

10.1.1 Supply the proof that $\boxed{\texttt{FlushBlanks(Ch)}} = F$ omitted in the analysis of `RemoveExtraBlanks`.

10.1.2 Suppose that `RemoveExtraBlanks` is given an input file that consists of more than one line. Explain in detail what will happen at the line boundaries.

10.1.3 Modify the design for `RemoveExtraBlanks` so that it preserves the line structure of the input, while still removing extra blanks on each line. Begin by writing a BNF description of the changed problem, using <line mark> for the line marker. Comment on how much of the analysis would have to be changed to handle the modified program.

10.1.4 Modify the design of `RemoveExtraBlanks` (to get `RemoveAllWords`) so that the action is reversed: write out only all the blanks and discard the words. For input string S, let the output from `RemoveExtraBlanks` be the string W, and that from `RemoveAllWords` be B. Explain why

$$\text{length}(S) \neq \text{length}(W) + \text{length}(B).$$

10.1.5 Prove that the implementation for

```
{insert blank if not last word}
```

could also be

```
IF Ch <> ' '
THEN
  WRITE(FileOut,' ')
```

10.1.6 A file of characters contains representations for numbers in base three, separated by blanks (perhaps several), and the numbers may have leading zero characters (perhaps several, or none). The problem is to reformat the file with single-blank spacers and no leading zeros (except that the number zero is to be represented by `0`).

a) Describe the problem using BNF.

b) Partition the input space according to strings that should be treated the same, and create test data for each such partition.

c) Give a design that solves the problem.

10.1.7 Given an input of two lines, the first a string of blanks, the second words and blanks, consider the blank string to specify a width by its length, and print each word from the second line centered in this width, on a separate line.

10.2 Sorting Word Strings

Preview

Sorting by words is more useful than by characters, but typical problems require a faster technique than those of Chapter 4. That technique is a repeated splitting and merging of strings.

There are few good reasons for sorting character strings, but there are many good reasons for sorting word strings. One example is in compiling the concordance of a book, that is, a list of words found in a book along with the page numbers where they appear. Such a concordance should be alphabetized, so sorting is required right there. However, just creating the list is a difficult job in which sorting plays a key role.

Consider a 400-page book with 500 words per page—a 200,000-word book. Suppose there were 3000 unique words in the book—how could they be found? One of the simplest ways would be to sort the entire book of words, then go through the sorted string and cast out duplicates, which will be all together after sorting. That sounds like a lot of work, and it is.

`SelectSort` requires n^2 reads and writes to sort a file of n characters. Suppose the average word size is 5 characters, with a blank between each word pair. `SelectSort` will require 200,000 $\times$ 200,000 $= 4 \times 10^{10}$ word reads and writes or 24×10^{10} character reads and writes. Since there are only 315,360 seconds in a year, even at a million reads and writes a second this would amount to over 9 months of execution, hardly practical. Another way to create the list would be to sort the words of the first page, cast out duplicates, then sort this list together with the next page, cast out duplicates again, and so on for 400 pages. At first there will be less than 500 words to sort for a page, but the majority of the 3000 unique words may occur within a few pages; once they do, there will be 3500 words to sort for each page. The worst case is to assume each sort is of 3500 words, done 400 times. The number of words to read and write for each page will be $3500 \times 3500 = 1.225 \times 10^7$, and for the book $3500 \times 3500 \times 400 = 4.9 \times 10^9$, or a total of 2.94×10^{10} character reads and writes, a factor of 8 better, but still over a month of execution.

The sorting strategy to be developed next requires only $n \times k$ reads to sort an n-character file, where k is the least integer such that

$$2^k \geq n.$$

For $n = 200{,}000$, the number of words in the book, the number of reads required by such a sort would be:

$$n \times k = 200{,}000 \times 18 = 3{,}600{,}000$$
(because $2^{18} = 262{,}144 > 200{,}000$).

Again assuming words to have five characters and to be separated by a space, sorting the words requires 21,600,000 character reads (a little more than 21 seconds). For simplicity and comparison with `SelectSort`, this new strategy will be developed to sort characters.

10.2.1 Sorting by Splitting and Merging

Two sorted strings can be merged into a single sorted string by repeatedly comparing their first characters, and moving the one that should come first to the end of the new string.

For example, with the two sorted strings:

```
cow
art
```

first **c** and **a** are compared and **a** is chosen to start the new string. Comparing **c** and **r** results in appending **c** to the end of the string. The steps of the completed merge are shown in the following table:

After step	Old strings		New string
0	cow	art	
1	cow	rt	a
2	ow	rt	ac
3	w	rt	aco
4	w	t	acor
5	w		acort
6			acortw

Thus, if a file can be split into sorted parts, called *runs*, it can be sorted by merging the runs.

An arbitrary string cannot easily be broken into two runs in one pass, but it can be broken into two strings, each of which contains several runs. By repeatedly splitting and merging a sorted string can be obtained. The process can be understood through an example. Consider the string

```
character
```

which contains four runs: **ch**, **ar**, **act**, **er**. By moving characters to two new strings and switching strings each time a run ends, the strings

```
ch·act
ar·er
```

are created. (The dot is not part of the string, but shows the break between runs.) Merging these character by character, the result is:

After step	Old strings		New string
0	chact	arer	
1	chact	rer	a
2	hact	rer	ac
3	act	rer	ach
4	ct	rer	acha
5	t	rer	achac
6	t	er	achacr
7	t	r	achacre
8	t		achacrer
9			achacrert

The merged string now has three runs: **ach**, **acr**, **ert**. Again splitting into two strings by runs gives

```
ach·ert
acr
```

and another merge produces

aacch·errt

Splitting a third time gives

aacch
errt

and a third merge yields the sorted string

aaccehrrt

Another way of merging is to observe the run boundaries and move all characters in a pair of runs before considering the next pair of runs. In the example, the first split:

ch·act
ar·er

would merge by runs to

achr·acert

and a second split and merge then gives the sorted string, more quickly than merging by characters.

The difference between merging by characters and merging by runs is profound. Merging by runs guarantees to reduce the number of runs to roughly half, because each pair of runs is turned into a single run. Merging by characters does not have this property. Therefore, even though the programming is more complex, sorting using splitting and merging by runs is now demonstrated.

A file to be sorted may already contain long runs, but it may not. The only runs certain to exist are those of length one—the individual characters. If the file is split on alternate characters, each split will have runs of length one. These can be merged to guarantee runs of length two. Splitting on every two characters and merging will give runs of length four, and so on. In this process the longer accidental runs that may appear in the files are ignored, but the procedure gains in regularity. It will be unnecessary to count if the (artificial) run boundaries are marked, for which we use the character **#**. An example of this process is shown below using an original string **wrcaot**:

Step	Split strings		Merged string
			w#r#c#a#o#t#
Split	**w#c#o#**	**r#a#t#**	
Merge			**rw#ac#ot#**
Split	**rw#ot#**	**ac#**	
Merge			**acrw#ot#**
Split	**acrw#**	**ot#**	
Merge			**acortw#**

The process stops when the text has only a single run. A procedure `MergeSort1` will now be designed using these ideas.

Design Part 2

```
PROCEDURE MergeSort1 (VAR Txt: TEXT);
  VAR
    Temp1, Temp2: TEXT;
    Sorted, Ch: CHAR;
  {Include PROCEDURE CopyOpen(VAR F1, F2: TEXT)
    Copy string in F1 into F2 without RESET or REWRITE;
    thus F1 must be ready to read and F2 ready to write,
    but the past strings of these files may not be empty}
  BEGIN {MergeSort1}
    {set Sorted to 'Y' if Txt is sorted
     and to 'N' otherwise};
    {divide the n characters in Txt into
     n runs of length 1 separated by '#'s};
    WHILE Sorted = 'N'
    DO
      BEGIN
        {split the n/k runs of length k in Txt
         into n/2k runs in each of Temp1 and Temp2};
        {merge the n/2k runs of length k in
         Temp1 and Temp2 into n/2k runs of length
         2k in Txt};
        {set Sorted to 'Y' if Txt is sorted
         and to 'N' otherwise}
      END
  END {MergeSort1}
```

10.2.2 Merging Runs

When the number of runs is odd, it cannot be split evenly into two files. For example, **rw#ac#ot#** splits into **rw#ot#** and **ac#**. Thus during the merge, after one split file has been completely read, the other may contain one more run, which must be copied into the merged file without any comparisons.

The part

```
{merge the n/2k runs of length k in
Temp1 and Temp2 into n/2k runs of length
2k in Txt}
```

will be designed as a procedure statement

```
MergeRuns(Temp1,Temp2,Txt)
```

with the declaration:

Design Part 2.1

```
PROCEDURE MergeRuns(VAR In1, In2, Result: TEXT);
  {merge the runs in In1 and In2
   into the runs in Result}
  VAR
    Ch1, Ch2: CHAR;
  BEGIN {MergeRuns}
    {initialize file parameters};
    WHILE NOT EOLN(In1) AND NOT EOLN(In2)
    DO
      BEGIN
        {merge runs from In1 and In2 into Result}
      END;
    {copy the last run of In1 or In2 (if any) into Result};
    {reset file parameters}
  END {MergeRuns}
```

During a merge, the characters contained in one of the two runs will be exhausted first. Thus t
remainder of either the first or second run needs to be copied to the resulting file.

Design Part 2.1.1

```
BEGIN {merge runs from In1 and In2 into Result}
  READ(In1,Ch1);
  READ(In2,Ch2);
  WHILE (Ch1 <> '#') AND (Ch2 <> '#')
  DO
    {copy the lesser of Ch1 and Ch2 to Result
     and get the next character from the
     corresponding file};
  {copy the rest of In1's run};
  {copy the rest of In2's run};
  WRITE(Result,'#')
END
```

Design Part 2.1.1.1

```
BEGIN {copy the lesser of Ch1 and Ch2
       to Result and get the next
       character from the corresponding file}
  IF Ch1 < Ch2
  THEN
    BEGIN
      WRITE(Result,Ch1);
      READ(In1,Ch1)
    END
  ELSE {Ch2 <= Ch1}
    BEGIN
      WRITE(Result,Ch2);
      READ(In2,Ch2)
    END
END
```

Design Part 2.1.1.2

```
BEGIN {copy the rest of In1's run}
  WHILE Ch1 <> '#'
  DO
    BEGIN
      WRITE(Result,Ch1);
      READ(In1,Ch1)
    END
END
```

Design Part 2.1.1.3

```
BEGIN {copy the rest of In2's run}
  WHILE Ch2 <> '#'
  DO
    BEGIN
      WRITE(Result,Ch2);
      READ(In2,Ch2)
    END
END
```

Design Part 2.1.2

```
BEGIN {copy the last run of In1 or In2 (if any) into Result}
  IF EOLN(In1) AND EOLN(In2)
  THEN
    WRITELN(Result)
  ELSE
    IF NOT EOLN(In1)
    THEN {copy the last run of In1 (if any) to Result}
      CopyOpen(In1,Result)
    ELSE {copy the last run of In2 (if any) to Result}
      CopyOpen(In2,Result)
END
```

Design Part 2.1.3

```
BEGIN {initialize file parameters}
  RESET(In1);
  RESET(In2);
  REWRITE(Result)
END
```

Design Part 2.1.4

```
BEGIN {reset file parameters}
  RESET(In1);
  RESET(In2);
  RESET(Result)
END
```

10.2.3 Splitting Runs

A procedure also hides the details of splitting one file into two; the part

```
{split the n/k runs of length k in Txt
into n/2k runs in each of Temp1 and Temp2}
```

will be designed as the procedure statement

```
SplitIntoRuns(Txt, Temp1, Temp2)
```

with the declaration:

Design Part 2.2

```
PROCEDURE SplitIntoRuns(VAR FileIn, Result1, Result2: TEXT);
  {split the runs in FileIn into runs in each
   of Result1 and Result2}
  VAR
    Switch, Ch: CHAR;
  BEGIN {SplitIntoRuns}
    RESET(FileIn);
    REWRITE(Result1);
    REWRITE(Result2);
    {copy runs alternately into Result1, Result2};
    WRITELN(Result1);
    RESET(Result1);
    WRITELN(Result2);
    RESET(Result2)
  END  {SplitIntoRuns}
```

SplitIntoRuns requires two character variables: Ch for copying character values and Switch for keeping track of the current copying file.

Design Part 2.2.1

```
BEGIN {copy runs alternately into Result1, Result2}
  Switch := '1';
  WHILE NOT EOLN(FileIn)
  DO
    BEGIN
      READ(FileIn,Ch);
      {copy Ch to Result1 if Switch='1'
       and to Result2 otherwise};
      {if end of run, flip switch to indicate other file}
END
```

Design Part 2.2.1.1

```
BEGIN {copy Ch to Result1 if Switch='1'
       and to Result2 otherwise}
  IF Switch = '1'
  THEN
    WRITE(Result1,Ch)
  ELSE
    WRITE(Result2,Ch)
END
```

Design Part 2.2.1.2

```
BEGIN {if end of run, flip switch
       to indicate other file}
  IF Ch = '#'
  THEN
    IF Switch = '1'
    THEN
      Switch := '2'
    ELSE
      Switch := '1'
END
```

10.2.4 Completing MergeSort1

The merged file is checked to determine if it is sorted in two different places in the program, so this check is also a natural candidate for a procedure statement:

```
{set Sorted to 'Y' if Txt is sorted
and to 'N' otherwise}
```

designed as

```
CheckIfSorted(Txt, Sorted)
```

CheckIfSorted sets Sorted to **Y** if there is only one run in the file (i.e., if the first # encountered immediately precedes the end of line).

Design Part 2.3

```
PROCEDURE CheckIfSorted(VAR FileIn: TEXT; VAR Sorted: CHAR);
  {set Sorted to 'Y' if FileIn contains
   one run and to 'N' otherwise}
  VAR
    Ch, EndRun: CHAR;
  BEGIN {CheckIfSorted}
    RESET(FileIn);
    IF EOLN(FileIn)
    THEN {empty file is sorted}
      Sorted := 'Y'
    ELSE
      {set Sorted to 'Y' if there is only
       one run in the file and to 'N' otherwise};
    RESET(FileIn)
  END {CheckIfSorted}
```

The local variable EndRun is set to **Y** as soon as the end of the first run is discovered. The first run must also be the last run if the file is really sorted. Thus, if the end-of-line condition is true after the first run in the file, the file is sorted.

Design Part 2.3.1

```
BEGIN {set Sorted to 'Y' if there is only
       one run in the file and to 'N' otherwise}
  EndRun := 'N';
  WHILE NOT EOLN(FileIn) AND (EndRun = 'N')
  DO
    BEGIN
      READ(FileIn,Ch);
      IF Ch = '#'
      THEN
        EndRun := 'Y'
    END;
  IF EOLN(FileIn) AND (EndRun = 'Y')
  THEN
    Sorted := 'Y'
  ELSE
    Sorted := 'N'
END
```

Finally, to start the whole process of splitting and merging, Txt is divided into one-character runs separated by # characters. A new character-valued variable, Ch, is added to the local declarations of MergeSort1 for the copying.

Design Part 2.4

```
BEGIN {divide the n characters in Txt into
       n runs of length l separated by '#'s}
  RESET(Txt);
  REWRITE(Templ);
  WHILE NOT EOLN(Txt)
  DO
    BEGIN
      READ(Txt,Ch);
      WRITE(Templ,Ch,'#')
    END;
  WRITELN(Templ);
  RESET(Templ);
  REWRITE(Txt);
  WHILE NOT EOLN(Templ)
  DO
    BEGIN
      READ(Templ,Ch);
      WRITE(Txt,Ch)
    END;
  WRITELN(Txt)
END
```

This completes the design of `MergeSort1`.

10.2.5 Exercises

10.2.1 Make up a set of test data that will structurally test each of the following program parts:

a) the procedure `MergeRuns`

b) the procedure `SplitIntoRuns`

c) the procedure `CheckIfSorted`

d) the procedure `MergeSort1`

10.2.2 Part d) of the previous exercise includes all the other parts. What advantage is there to testing procedures separately if they will eventually be tested together?

10.2.3 Devise a partition of the input space for a file marked off into runs, and use it to partition test `SplitIntoRuns` and `CheckIfSorted`.

10.2.4 Give a formal proof that `CheckIfSorted` does what it is supposed to do.

10.2.5 Consider splitting and merging runs to and from three files instead of two.

a) Describe how a procedure similar to `MergeSort1` would sort using three files.

b) Explain why such a sort would be faster, using an example.

c) Repeat b) for four splitting files.

10.2.6 Discover a fast way to reverse a file by considering an example file that is in the reverse of sorted order, and examining its sorting by `MergeSort1`. From this example work out a design for reversing a file by splitting and merging.

10.2.7 Modify `MergeSort1` to take advantage of runs already in the original string and of runs accidentally created on each merge pass. How much would this help on randomly ordered strings?

10.2.8 Modify `MergeSort1` so that its action is exactly the same, but isolate the information that it is sorting *characters*. Do this by replacing all direct character operations, e.g.,

```
IF Ch1 < Ch2 ...
```

by procedure statements, e.g.,

```
BEGIN
  Compare(Ch1, Ch2, Result);
  IF Result = '<' ...
END
```

The text of these procedures would be trivial. (This kind of work, changing a program to behave no differently, but to use a structure that is adaptable to future changes, is called *preventive maintenance*. See the following two exercises.)

10.2.9 Modify `MergeSort1` so that it sorts words instead of characters. Use the scheme of the previous exercise, but now write the procedures you've inserted with appropriate `TEXT` arguments for words.

10.2.10 Modify `MergeSort1` so that it sorts lines instead of characters. (See the previous exercise.)

10.3 An Improved MergeSort

Preview

`MergeSort1` is a much better sorting algorithm than `SelectSort`, but it can be further improved to take advantage of accidental runs.

Each time `MergeRuns` executes, the size of each run in `Txt` is doubled. The following table shows the sizes of the runs after each execution of `MergeRuns`.

Merge	Run Size
1	2
2	4
3	8
4	16
5	32
k	2^k

`Txt` is sorted when only a single run remains. For n characters, this will occur when k merges have been done, and $2^k \geq n$. Since each merge must read and write all n characters of the file and k merges are needed, the number of reads and writes performed by this program is $n \times \log_2 n$. It is interesting to compare `MergeSort1` with `SelectSort`, which sorted a file of size m in $m \times m$ reads and writes. To make a fair comparison, assume `SelectSort` works on a file of length n and `MergeSort1` works on a file of length $2n$. (`MergeSort1` initially doubles the length of the file with its # run markers. These marks decrease in number as the sorting proceeds, but taking the length to be $2n$ is more than fair to `Selectsort`.) The comparison is:

Length of file	Number of reads and writes	
	`SelectSort`	`MergeSort1`
n	n^2	$2n \times \log_2 2n$
10	100	86
100	10,000	1,520
1000	1,000,000	22,000
10000	100,000,000	840,000

`MergeSort1` could be improved by taking advantage of the parts of a file that are already sorted. Instead of inserting arbitrary run markers, the actual runs might be used. With fewer, longer runs, fewer calls to `MergeRuns` are needed. Of course if the file is in descending order, all runs have length 1, but this will rarely be the case. Run markers are unnecessary since one run ends and another begins wherever the last character read is greater than the current character.

An example of the improved sort is shown below. The dots that mark runs are not actually present.

Step	Split strings		Merged string
			wrcaot
Split	**w·c**	**r·aot**	
Merge			**rw·acot**
Split	**rw**	**acot**	
Merge			**acortw**

The first split kept the run **aot** together because the letters were already in ascending order. Only two merges are needed instead of three.

It is difficult to analyze the effect of using the natural runs in the merged files, because how much is gained depends on the way in which runs deviate from the worst case of a reverse-ordered file. However, even in the worst case the number of read and write operations is reduced from $2n \times \log_2 2n$ to $n \times \log_2 n$, which is better than a factor of six when $n = 10{,}000$.

The design for an improved `MergeSort2` using actual runs is nearly the same as that for `MergeSort1`, except that the `#` markers need not be inserted:

Design Part 3

```
PROCEDURE MergeSort2 (VAR Txt: TEXT);
  VAR
    Temp1, Temp2: TEXT;
    Sorted: CHAR;
  BEGIN {MergeSort2}
    CheckIfSorted(Txt,Sorted);
    WHILE Sorted = 'N'
    DO
      BEGIN
        SplitIntoRuns(Txt,Temp1,Temp2);
        MergeRuns(Temp1,Temp2,Txt);
        CheckIfSorted(Txt,Sorted)
      END
  END {MergeSort2}
```

Each design part of `MergeSort1` can be reused, with the changes below (numbered to correspond to the numbers in Section 10.2).

Design Part 2.2.1 must be modified to provide a new method for copying runs alternately into `Result1` and `Result2`. By passing a two-character window (`LastCh` and `Ch`) over `FileIn`, the end of one run and the beginning of another occurs when the value of `Ch` precedes the value of `LastCh`.

Design Part 3.2.1

```
BEGIN {copy runs alternately into Result1, Result2}
  Switch := '1';
  Ch := '#';
  WHILE NOT EOLN(FileIn)
  DO
    BEGIN
      LastCh := Ch;
      READ(FileIn,Ch);
      {if end of run, flip switch to indicate other file};
      {copy Ch to Result1 if Switch='1'
       and to Result2 otherwise}
    END
END
```

Ch is assigned the arbitrary value # before entering the WHILE statement so that the window exists even at the first character.

Runs are no longer explicitly marked, so Design Part 2.2.1.2 must be modified as shown below.

Design Part 3.2.1.2

```
BEGIN {if end of run, flip switch to indicate other file}
  IF Ch < LastCh
  THEN
    IF Switch = '1'
    THEN
      Switch := '2'
    ELSE
      Switch := '1'
END
```

Merging by runs is more complex than splitting when the runs are not explicitly marked. If the first character of the next run of a file is smaller than the remaining characters in the current run of another file, the run boundaries could be violated. The solution to this problem is to continue reading characters from the same run as long as possible. If the last character added to the merged run is less than either of the characters at the heads of the runs to be merged, then both runs still have characters to process. However, if this character is greater than one of the characters at the head of a run but not the other, then the first run has ended but the second has more characters to process. As an example, consider the details of the first merge operation above:

Step	Split strings		Merged string
			wrcaot
Split	w·c	r·aot	
Merge			
1	w·c	aot	r
2	c	aot	rw
3	c	ot	rw·a
4		ot	rw·ac
...			

After step 1 of the merge, the r added to the merged string is less than the w on the first split string, so the run there has not ended; but, r is greater than the a on the second split string, so that run has ended.

A more systematic analysis of this situation can be accomplished using a table to analyze the relations between the last character added to the merged run *m*, the next character *s* in Temp1, and the next character *t* in Temp2.

	$s < t$	$s \geq t$
$m \leq s$ $m \leq t$	$t > s \geq m$ both runs continue next char from Temp1	$s \geq t \geq m$ both runs continue next char from Temp2
$m \leq s$ $m > t$	$t > s \geq m > t$ impossible condition	$s \geq m > t$ Temp2's run finished next char from Temp1
$m > s$ $m \leq t$	$t \geq m > s$ Temp1's run finished next char from Temp2	$s \geq t \geq m > s$ impossible condition
$m > s$ $m > t$	$m > t > s$ both runs finished next char from Temp1	$m > s \geq t$ both runs finished next char from Temp2

This analysis leads to an altered design for `MergeRuns`. The most obvious change is that `Ch` must be initialized to the smaller of the first characters in `In1` and `In2`, so that the above tests can be applied.

Design Part 3.1

```
PROCEDURE MergeRuns(VAR In1, In2, Result: TEXT);
  {merge the runs in In1 and In2
   into the runs in Result}
  VAR
    Ch, Ch1, Ch2, Choice: CHAR;
  BEGIN {MergeRuns}
    {initialize file parameters};
    READ(In1,Ch1);
    READ(In2,Ch2);
    {if neither In1 nor In2 are empty, Ch := min(Ch1,Ch2)};
    WHILE NOT EOF(In1) AND NOT EOF(In2)
    DO
      BEGIN {merge runs from In1 and In2 into Result}
        {set Choice to '1' to read from In1,
        '2' to read from In2};
        {replace Ch with Ch1 or Ch2 depending on the value
         of Choice and get a new value for Ch1 or Ch2}
      END;
    {copy the last run of In1 or In2 into Result};
    {reset file parameters}
  END {MergeRuns}
```

Design Part 3.1.5

```
BEGIN {if neither In1 nor In2 are empty, Ch := min(Ch1,Ch2)}
  IF NOT EOF(In1) AND NOT EOF(In2)
  THEN
    {Ch := min(Ch1,Ch2)}
    IF Ch1 < Ch2
    THEN
      Ch := Ch1
    ELSE
      Ch := Ch2
END
```

The choice of where to read the next character is made according to the analysis above:

Design Part 3.1.1

```
BEGIN {set Choice to '1' to read from In1,
       '2' to read from In2}
  IF (Ch <= Ch1) OR (Ch <= Ch2)
  THEN {current run can be preserved}
    IF (Ch2 > Ch1) AND (Ch1 >= Ch)
      OR (Ch1 >= Ch) AND (Ch > Ch2)
    THEN
      Choice := '1'
    ELSE
      Choice := '2'
  ELSE {start new run with lesser character}
    IF Ch1 < Ch2
    THEN
      Choice := '1'
    ELSE
      Choice := '2'
END
```

Finally, the last value that was written to `Result` becomes the new value of `Ch`. Whichever of `Ch1` or `Ch2` had its value copied to `Ch` must be replaced with a new value from `In1` or `In2` appropriately.

Design Part 3.1.2

```
BEGIN {replace Ch with Ch1 or Ch2 depending on the value
       of Choice and get a new value for either Ch1 or Ch2}
  IF Choice = '2'
  THEN
    BEGIN
      Ch := Ch2;
      WRITE(Result,Ch2);
      READ(In2,Ch2)
    END
  ELSE
    BEGIN
      Ch := Ch1;
      WRITE(Result,Ch1);
      READ(In1,Ch1)
    END
END
```

Either `Ch1` or `Ch2` (the one whose value was not chosen as the value of `Ch`) and any remaining characters that follow them in `In1` or `In2` respectively must be added to `Result`.

Design Part 3.1.3

```
BEGIN {copy the last run of In1 or In2 into Result}
  IF NOT EOF(In1)
  THEN {copy the last run of In1 (if any) to Result}
    BEGIN
      WRITE(Result,Ch1);
      CopyOpen(In1,Result)
    END
  ELSE {copy the last run of In2 (if any) to Result}
    BEGIN
      WRITE(Result,Ch2);
      CopyOpen(In2,Result)
    END
END;
```

All that remains is to modify CheckIfSorted to reflect the fact that # is no longer used to mark runs.

Design Part 3.3

```
PROCEDURE CheckIfSorted (VAR FileIn: TEXT; VAR Sorted: CHAR);
  {set Sorted to 'Y' if FileIn contains
   one run and to 'N' otherwise}
  VAR
    Ch, LastCh: CHAR;
  BEGIN {CheckIfSorted}
    RESET(FileIn);
    Sorted := 'Y';
    IF NOT EOLN(FileIn)
    THEN
      READ(FileIn,LastCh);
    WHILE NOT EOLN(FileIn) AND (Sorted = 'Y')
    DO
      BEGIN
        READ(FileIn,Ch);
        IF Ch < LastCh
        THEN
          Sorted := 'N';
        LastCh := Ch
      END;
    RESET(FileIn)
  END {CheckIfSorted}
```

The complete design for MergeSort2 may now be assembled using Design Parts that begin with 3 where they have been given, and otherwise those that begin with 2 from MergeSort1.

10.3.1 Exercises

10.3.1 In MergeSort1 and MergeSort2 there are a number of differences between Design Parts 2.1.2 and 3.1.3, that play a very similar role in the different versions of MergeRuns. Account for each difference.

10.3.2 In the analysis of merging natural runs without boundary markers, a table was given to determine whether runs are ended or not, based on the values of characters heading the split files, and the character just merged. Prove each of the statements in this table that refer to runs being finished.

10.4 Chapter Summary

Word strings offer a substantial challenge for the programmer. Although this is a very simple string and substring structure, there are problems in processing it that require all the power of good design and analysis to solve correctly.

The problem of removing extra blanks from a file was used to demonstrate problem description using BNF, and the method of partition testing based on such a description. The proof that the design does what it is supposed to do is the most difficult seen so far in this text.

Sorting by splitting and merging files is a far more efficient technique than those investigated in Chapter 4. It sorts a file of length n using roughly $n \times \log_2 n$ read and write operations.

CHAPTER 11

PROGRAMMING WITH RECURSION

Chapter Preview

Recursion is a problem-solving approach in which a problem to be solved is reduced to a simpler problem of the same type. The complete problem solution takes the form of a recursive procedure, which makes the necessary reduction, then invokes itself to solve the simpler problem.

11.1 Recursive Procedures in CF Pascal

Preview

Procedures in CF Pascal can include procedure statements naming themselves. For some problems, recursion is the natural and comprehensible approach, superior to an iterative solution.

New ideas: recursion and its implementation in CF Pascal

11.1.1 Problem Solving with Recursion

The family-of-problems method presented in Chapter 4 leads to recursive solutions. As illustrated in the `IFSort` and `MinSort` families of programs, this approach sorted longer strings using previously developed sorts for short strings. For example, in the `MinSort` family of programs, `MinSort4` (which sorted 4-strings) could be described in terms of `MinSort3`; `MinSort5` could be described in terms of `MinSort4`; etc. The solution to sorting an m-string is to use an (m-1)-string sort.

This is precisely the concept in *SelectSort*; given an m-string to sort, the minimum character is removed. But this leaves an (m-1)-string to sort, which *SelectSort* completes by continuing the process, getting an (m-2)-string, ..., 2-string, 1-string to sort in turn. The final problem, sorting a 1-string, has a trivial solution.

This problem-solving strategy finds a way to repeatedly reduce a problem to a smaller problem of the same form, until at last a problem is reached which can be solved directly. Suppose a procedure can be written to solve a problem for any member of such a family, with the problem data in the procedure parameter. Then, this procedure should be able to call itself with a parameter containing the data of a smaller member of the family. For example, imagine a procedure to sort a string in file `F1` to `OUTPUT`:

```
PROCEDURE RSort(VAR F1: TEXT);
  VAR
    Min: CHAR;
    F2: TEXT;
  BEGIN {RSort}
    RESET(F1);
    IF NOT(EOLN(F1))
    THEN
      BEGIN
        REWRITE(F2);
        {Place the minimum character of F1 in Min,
         and copy the other characters to F2};
        WRITE(Min);
        {Copy F2 back to F1};
        {***}
      END
  END  {RSort}
```

At the place marked {***}, F1 holds a string one character shorter than it did when the procedure was begun, and the first character of the sort has been written. That is, the problem has been reduced to sorting a shorter string, in exactly the same format for which RSort was designed. Thus {***} can be replaced by the procedure statement:

```
RSort(F1)
```

In the case of an empty file, there is nothing to write to OUTPUT, and RSort does exactly that, because of its test for EOLN.

Now suppose RSort appears in a procedure statement:

```
PROGRAM DoRSort(INPUT,OUTPUT);
  VAR
    Raw: TEXT;
  {Include PROCEDURE RSort(VAR F1: TEXT)}
  BEGIN {DoRSort}
    REWRITE(Raw);
    {Copy a line of INPUT to Raw};
    WRITELN('Sorted output is:');
    RSort(Raw);
    WRITELN
  END. {DoRSort}
```

There appears to be something magical about this sorting program. RSort seems to process only one character, not to sort at all. But it will be executed over and over within itself, each time processing one character, until finally the parameter file becomes empty. Every character in the file will be added to OUTPUT, and in sorted order.

11.1.2 Recursive Procedure Execution

When a procedure declaration includes a procedure statement with its own name—a recursive call—nothing very special happens. Newly activated parameters and variables suppress any references to any previously activated namesakes, which just happen to include their own previously activated instances. Neither does anything special happen when such a procedure completes execution. The latest activations of the procedure variables disappear, leaving the previous namesakes accessible, and the statement following the procedure statement is executed.

Because each invocation of a recursive procedure uses the same local-variable names (and often the same parameter names), there is potential for confusing the separate values attached to these names. Keeping track of values requires care, but is not much different than the care required to handle scope confusions that occur without recursion. Consider the recursive procedure in the following program:

```
PROGRAM RunReverse(INPUT,OUTPUT);
  PROCEDURE Reverse(VAR File1: TEXT);
    VAR
      Ch: CHAR;
    BEGIN {Reverse}
      IF NOT(EOLN(File1))
      THEN
        BEGIN
          READ(File1, Ch);
          Reverse(File1);
          WRITE(Ch)
        END
    END; {Reverse}
  BEGIN {RunReverse}
    Reverse(INPUT);
    WRITELN
  END.  {RunReverse}
```

Execution

```
INPUT :AB
OUTPUT:BA
```

In this execution `Reverse` is invoked three times, the first time in the procedure statement of the program, the last two times in the procedure statement of the procedure. Each such procedure statement causes the execution of the procedure block, beginning with the `VAR` declaration. Each such declaration defines a new copy of variable `Ch`, and the Pascal machine holds and hides the previous variable of the same name.

The first two executions include execution of a procedure statement using `Reverse` again, but the third execution is completed without any effect because the IF condition fails. At completion of each procedure statement, two things happen, namely (1) the variable `Ch` for that procedure execution disappears, exposing a hidden `Ch` (for the two calls within `Reverse`), and (2) the next statement is executed. In the inner two executions, this next statement is `WRITE(Ch)`, and in the outer execution the program terminates.

The execution table below shows in detail what happens.

Execution Table for RunReverse

After execution of	INPUT	OUTPUT	EOLN(File1)	Ch in execution 1	2	3
	AB					
PROGRAM RunReverse...	AB/	_				
BEGIN						
Reverse(INPUT)						
BEGIN						
VAR Ch: CHAR				?		
IF NOT(EOLN(...			FALSE			
BEGIN						
READ(File1,Ch)	AB/			A		
Reverse(File1)						
BEGIN						
VAR Ch: CHAR					?	
IF NOT(EOLN(...			FALSE			
BEGIN						
READ(File1,Ch)	AB/				B	
Reverse(File1)						
BEGIN						
VAR Ch: CHAR						?
IF NOT(EOLN(...			TRUE			
END						
WRITE(Ch)		B_				
END						
END						
WRITE(Ch)		BA_				
END						
END						
WRITELN		BA/_				
END.	AB	BA				

The next example reverses the statements within the recursive procedure: first it writes, then makes the recursive call. The result is to copy in the original, not reversed, order.

```
PROGRAM RunCopy(INPUT,OUTPUT);
  PROCEDURE Copy(VAR File1: TEXT);
    VAR
      Ch: CHAR;
    BEGIN  {Copy}
      IF NOT(EOLN(File1))
      THEN
        BEGIN
          READ(File1, Ch);
          WRITE(Ch);
          Copy(File1)
        END
    END;  {Copy}
  BEGIN {RunCopy}
    Copy(INPUT);
    WRITELN
  END.  {RunCopy}
```

Execution

```
INPUT :AB
OUTPUT:AB
```

In the iterative version of copying, each character is moved from the input file through a program variable to the output file. The recursive version doesn't work this way at all; instead, each input character is held in a different local variable, and all exist in the Pascal machine at the same time. These characters can be recovered, as in the next example, a program which first copies the input characters, then adds them in reversed order.

```
PROGRAM RunPalindrome(INPUT,OUTPUT);
  PROCEDURE Palindrome(VAR File1: TEXT);
    VAR
      Ch: CHAR;
    BEGIN {Palindrome}
      IF NOT(EOLN(File1))
      THEN
        BEGIN
          READ(File1, Ch);
          WRITE(Ch);
          Palindrome(File1);
          WRITE(Ch)
        END
    END;  {Palindrome}
  BEGIN {RunPalindrome}
    Palindrome(INPUT);
    WRITELN
  END.  {RunPalindrome}
```

Executions

```
INPUT :AB
OUTPUT:ABBA
INPUT :123456789
OUTPUT:123456789987654321
```

11.1.3 Exercises

11.1.1 In the program `DoRSort` that begins this chapter, why can't the WRITELN statements be put inside the procedure `RSort`?

11.1.2 Create an execution table for the first sample execution of `RunPalindrome`.

11.1.3 Make up data that will structurally test `RunPalindrome`.

11.1.4 The danger in recursive programming is that the recursion will be unending. It is essential within a recursive procedure to (A) test for termination, and (B) reduce the problem to a strictly simpler one before making the recursive call. Consider the program:

```
{Include PROCEDURE NextDigit(VAR D1: CHAR);
  D1 is replaced by its successor in the
  sequence 0123456789012...}
PROCEDURE CountUp(VAR Now: CHAR);
  BEGIN {CountUp}
    IF Now < '9'
    THEN
      BEGIN
        WRITELN(Now);
        NextDigit(Now);
        CountUp(Now)
      END
  END  {CountUp}
```

a) Write out the execution table starting with a state in which Now is initially 3.

b) Why is the test in general inadequate (see (A) above)? Suppose the calls on NextDigit and CountUp were interchanged. Why then has the problem not been properly reduced (see (B) above)?

11.1.5 "Tail recursion" occurs when the unit of work is done *before* the recursive call, as in RunCopy, so the execution sequence is:

work/call/work/call/ ... /work/call/return/ ... /return/return.

The opposite is "initial recursion" where the work is done *after* the recursive call, as in RunReverse, with execution sequence:

call/call/ ... /call/work/return/ ... /work/return/work/return.

a) Show where the tests for termination should appear in each of these sequences.

b) Which kind of recursion does RunPalindrome use?

11.2 Sorting and Reversing with Recursion

Preview

Recursion often leads to program designs that are simple and elegant. The idea behind fast sorting using splitting and merging can be applied recursively.

New ideas: recursive sorting, recursive reverse

11.2.1 Recursive Sort

RSort sorts using the idea of repeated selection, iteratively expressed in SelectSort. However, MergeSort is a much faster iteration sort than SelectSort. Is there a recursive counterpart to MergeSort? To apply recursion effectively, the string to be sorted must be divided, say in half. If the halves are separately sorted, they can be merged to give the desired result, as indicated by the example:

Given:

†**character**†

Split into half strings:

†**chara**† †**cter**†

Sort half strings:

†**aacgr**† †**cert**†

Merge half string into sorted string:

†**aaccehrrt**†

The recursion will occur in the second step: the same procedure can be used to sort the half strings that was applied to the whole.

It remains only to find a way to split a file into halves. One way is to copy every other character of the first file into a "counting file," then use this counting file to measure out the first half of the original file into another file, then copy the rest of the original into yet another file. But the counting file itself is half the original, just not the *first* half. It is an essential feature of sorting that it doesn't matter how the original file is scrambled, so long as no characters are lost, and so taking alternate characters for the split will do, as this example shows:

Given:

†**character**†

Split odd and even characters:

†**caatr**† †**hrce**†

Sort half strings:

†**aacrt**† †**cehr**†

Merge half strings into sorted string:

†**aaccehrrt**†

The half strings are different, but the final result is the same as before.

Each recursion will be sorting a file that is no more than half the length of the one it began on. Eventually this division will produce a file of one character, which is already sorted. Thus the recursion is ended by checking the file length.

These ideas lead to a design for a fast recursive sort:

Design Part 1

```
PROCEDURE RecursiveSort(VAR F1: TEXT);
  VAR
    F2, F3: TEXT;
    Ch: CHAR;
  {PROCEDURE Split(VAR F1, F2, F3: TEXT)
    Split F1 between F2,F3}
  {PROCEDURE Merge(VAR F1, F2, F3: TEXT)
    Merge F2,F3 into F1}
  BEGIN {RecursiveSort}
    RESET(F1);
    IF NOT(EOLN(F1))
    THEN
      BEGIN
        READ(F1, Ch);
        IF NOT(EOLN(F1))
        THEN {File has at least 2 characters}
          BEGIN
            Split(F1, F2, F3);
            RecursiveSort(F2);
            RecursiveSort(F3);
            Merge(F1, F2, F3)
          END
      END
  END {RecursiveSort}
```

Design Part 1.1

```
PROCEDURE Split(VAR F1, F2, F3: TEXT);
  {Split F1 between F2, F3}
  VAR
    Ch, Switch: CHAR;
  BEGIN {Split}
    RESET(F1);
    REWRITE(F2);
    REWRITE(F3);
    {Copy F1 alternatively to F2, F3};
    WRITELN(F2);
    WRITELN(F3)
  END {Split}
```

Design Part 1.1.1

```
BEGIN {Copy F1 alternatively to F2, F3}
  Switch := '2';
  WHILE NOT(EOLN(F1))
  DO
    BEGIN
      READ(F1, Ch);
      IF Switch = '2'
      THEN
        BEGIN
          WRITE(F2, Ch);
          Switch := '3'
        END
      ELSE
        BEGIN
          WRITE(F3, Ch);
          Switch := '2'
        END
    END
END
```

Design Part 1.2

```
PROCEDURE Merge(VAR F1, F2, F3: TEXT);
  {Merge F2, F3 in sorted order into F1}
  VAR
    Ch2, Ch3: CHAR;
  BEGIN {Merge}
    RESET(F2);
    RESET(F3);
    REWRITE(F1);
    READ(F2, Ch2);
    READ(F3, Ch3);
    WHILE (NOT(EOF(F2))) AND (NOT(EOF(F3)))
    DO
      IF Ch2 < Ch3
      THEN
        BEGIN
          WRITE(F1, Ch2);
          READ(F2, Ch2)
        END
      ELSE
        BEGIN
          WRITE(F1, Ch3);
          READ(F3, Ch3)
        END;
    {Copy rest of F2 to F1};
    {Copy rest of F3 to F1};
    WRITELN(F1)
  END {Merge}
```

Design Part 1.2.1

```
BEGIN {Copy rest of F2 to F1}
  WHILE NOT(EOF(F2))
  DO
    BEGIN
      WRITE(F1, Ch2);
      READ(F2, Ch2)
    END
END
```

Design Part 1.2.2

```
BEGIN {Copy rest of F3 to F1}
  WHILE NOT(EOF(F3))
  DO
    BEGIN
      WRITE(F1, Ch3);
      READ(F3, Ch3)
    END
END
```

The recursive solution is cleaner than the iterative one, because it need not identify or preserve run boundaries.

11.2.2 Recursive Reverse

Recursion can be used to reverse a string in a similar way, as indicated by the example:

Given:

†**character**†

Split string into first half, second half:

†**chara**†, †**cter**†

Reverse half strings:

†**arahc**†, †**retc**†

Interchange reversed half strings:

†**retc**†, †**arahc**†

Combine interchanged reversed half string into reversed string:

†**retcarahc**†

Again, the awkward half split can be avoided. Suppose the file were split by alternate characters, as for sorting. If these parts were each reversed, then woven back together alternately, the result would be the reversal of the original string, as in the example:

Given:

†**character**†

Split string into odds, evens:

†**caatr**†, †**hrce**†

Reverse half strings:

†**rtaac**†, †**ecrh**†

Interweave half strings:

†**retcarahc**†

The example shows that care is required when interweaving the reversed strings: the first character must come from the split that received the last character of the original. These ideas lead to a design for fast recursive reversal:

Design Part 2

```
PROCEDURE RecursiveReverse(VAR F1: TEXT);
  VAR
    F2, F3: TEXT;
    Ch, Last: CHAR;
  BEGIN {RecursiveReverse}
    RESET(F1);
    IF NOT(EOLN(F1))
    THEN
      BEGIN
        READ(F1, Ch);
        IF NOT(EOLN(F1))
        THEN {File contains at least two characters}
          BEGIN
            {Split F1 into F2 and F3 and record which
             receives the final character in Last};
            RecursiveReverse(F2);
            RecursiveReverse(F3);
            {Interweave F2 and F3 into F1 beginning
             according to what is recorded in Last}
          END
  END  {RecursiveReverse}
```

The remaining parts of the design are similar to those in `RecursiveSort`.

11.2.3 Exercises

11.2.1 The procedure `Reverse` of Section 11.1.2 does string reversal.

a) Work out the number of read/writes needed to reverse a file of length n.

b) Similarly analyze `RecursiveReverse` and compare.

c) Contrast these two reversing routines by where they hold the characters before releasing them

d) Discuss using the idea of `Reverse` to do sorting.

11.2.2 Make up a set of data that will structurally test each of the following parts of `RecursiveSort`.

a) the procedure `Split`

b) the procedure `Merge`

c) the complete procedure `RecursiveSort`

11.2.3 Create an execution table for `RecursiveSort` with input string †**character**†.

11.2.4 `RecursiveSort` may carry out unnecessary recursion by failing to recognize files which are already sorted. Modify it to check whether `F1` is sorted before using the recursive procedure statements.

11.2.5 Complete the design of `RecursiveReverse`.

11.2.6 In `RecursiveReverse`, explain why the variable `Last` must be a local of the procedure, rather than a global in a program that calls the procedure. Is the same true for `F2` and `F3`?

11.3 Solving Problems with Recursion

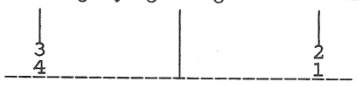
Preview

Recursion is a general problem-solving technique which can be helpful in many situations. The classic Towers of Hanoi problem has an elegant recursive solution.

New ideas: Towers of Hanoi.

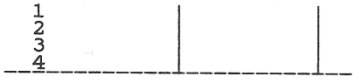
11.3.1 The Towers of Hanoi

The Towers of Hanoi is an ancient puzzle which uses a board with three pegs and a set of disks of distinct sizes, each with a hole which permits it to be slid onto a peg. A set of disks stacked on a peg is called a *tower* if no disk rests on a smaller disk; otherwise the set is called a *rubble*. By this definition there is a tower on a peg with no disks. It is convenient to represent a disk with a number, a larger number signifying a larger disk. In this form, here is a representation of a board:



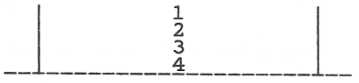
The first and second pegs hold towers; the third peg holds a rubble.

The Towers of Hanoi begins with one tower on the board. The puzzle is solved if this tower can be transferred to another peg by moving only one disk at a time, never creating a rubble. That is, at every step, the board must contain three towers. If the number of disks is n, we say it is an n-disk puzzle. For example, a 4-disk puzzle begins with the board:



T1

and would be solved with the board



T2

(and the proper moves to make the transition).

The Towers of Hanoi puzzle is a worthy adversary for the intuition. It would be an important intermediate step to get disk 4 from the first peg to the second peg. That could be accomplished from the position:



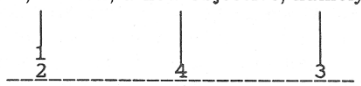
T3

with the move:



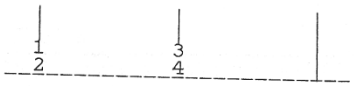
T4

This suggests, in turn, a new objective, namely to create the board:



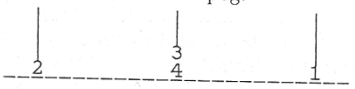
T5

in which disk 3 can be placed on top of disk 4:



T6

then disk 1 moved to the third peg:



T7

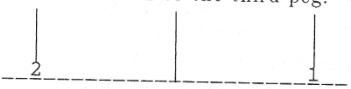
and the puzzle finished by moving disk 2, then disk 1 to the second peg.

The board T5 seems to be a good objective, but it may be hard to realize. However, the boards T6, T7, T2 were created step by step from T5, and provide a basic pattern that can be generalized and expanded.

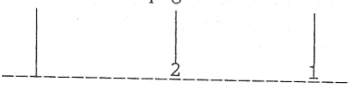
In order to see how to move a subtower from one peg to another, consider moving a 2-disk tower from the first peg to the second peg:



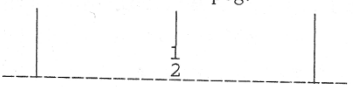
As in T6, first move disk 1 to the third peg:



then disk 2 to the second peg:



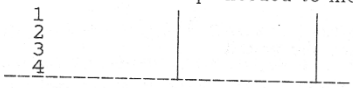
and finally disk 1 to the second peg:



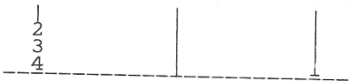
Here are several observations about this move:

1. The entire move takes three steps.
2. The first disk goes to the peg not intended as the final destination for the tower.
3. The move of tower 1/2 can be a step in moving subtower 1/2/3, by regarding the latter as a two-disk problem where one disk is the composite (1/2), the other is 3.

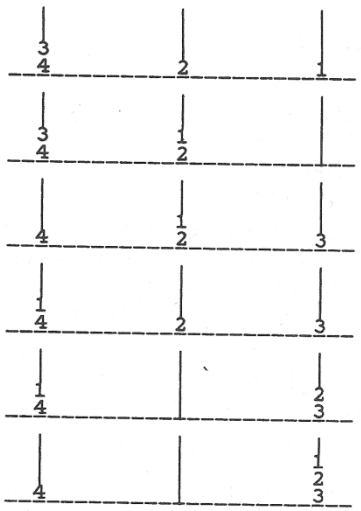
Observation 3 returns to the 4-disk problem, and how to get from T1 to T3, namely how to move subtower 1/2/3 from the first peg to the third peg. Regard the subtower 1/2/3 as a 2-tower (1/2)/3 with composite disk 1/2. By observation 2, (1/2) must not go to the destination. That is, we want to move 1/2 to the second peg. But in order to do that, its first disk (disk 1) must go to the nondestination, namely the third peg (the destination for the subtower 1/2/3, but not 1/2). This discussion has outlined the steps needed to move from T1 to T3:



T1

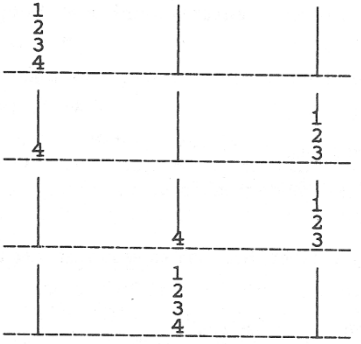


T8



In this process the composite disk (1/2) is moved twice, each move requiring 3 steps, and disk 3 is moved once, so there are $3 + 3 + 1 = 7$ steps in moving the 3-tower.

A final observation generalizes the solution for a 4-disk puzzle to that for an n-disk puzzle. The moves from T1 to T3 to T4 to T2 may be thought of as a 2-disk move with composite disk (1/2/3) and disk 4:



Therefore, an n-disk puzzle can be solved by a 2-disk move with composite disk (1/2.../n-1) and disk n.

11.3.2 Mathematical Induction

Although the Towers of Hanoi puzzle can be solved in a few steps with a few disks, it is very difficult for more disks. The legend has it that a 64-disk puzzle, the disks made of gold on diamond spindles, is worked on day and night by specially trained monks. When they solve it, the world will end. When will that be? The principle of solution is simple, but the number of steps is immense. If the 64-disk puzzle took 16 times as many steps as the 4-disk puzzle (because it is that many more disks), it would be tedious, but could be done at one sitting. If it took $16^2 = 256$ times as many steps, the monks would be bored, but the world would soon end. Could it require more steps?

The number of steps required to solve an n-disk puzzle in the way indicated above can be calculated. For a few cases:

Number of Disks n	Steps S_n
1	1
2	3
3	$2(3) + 1 = 7$
4	$2(7) + 1 = 15$
5	$2(15) + 1 = 31$
n	$2(S_n) + 1$

Each S_n is obtained as the number of steps required to move a composite (1/2/.../n-1) disk twice and disk n once. The number of steps in the composite move can be obtained from the previous line of the table. In fact even S_1 and S_2 can be described by the same formula, since $1 = 2(0) + 1$, and $3 = 2(1) + 1$. Do the number of steps 1, 3, 7, 15, 31, ... have a pattern? Each is one less than the successive powers of 2, as seen in a table with an additional column:

n (Disks)	S_n (Steps)	2^n
1	1	2
2	3	4
3	7	8
4	15	16
5	31	32

But what of the general case? Is

$S_n = 2^n - 1$?

Mathematical induction is a method for proving formulas of this kind.

Let a Boolean expression B(n) be written in terms of some natural number n. Suppose that for one natural number n_0, the expression holds:

H1: $B(n_0) = TRUE$,

and that for every $n \geq n_0$, if B(n) holds, then so does B(n+1):

H2: if $B(n) = TRUE$ then $B(n+1) = TRUE$, all $n \geq n_0$.

Then it follows that in fact B(k) holds for every natural number $k \geq n_0$:

C: $B(k) = TRUE$, all $k \geq n_0$.

This principle of mathematical induction is intuitively correct, because to establish B(64), given that B(1), proceed as follows:

Identification	Conclusion	Reason
C0	B(1)	by *H1*
C1	B(2)	by *C0* and *H2*
C2	B(3)	by *C1* and *H2*
...		
C64	B(64)	by *C63* and *H2*

H1 is called the *initial hypothesis* (or *base case*), and *H2* is called the *induction hypothesis*. *C* is called the *induction conclusion*. Proving *H1* and *H2* in order to conclude *C* is called a *proof by induction*.

For the Towers of Hanoi, B(n) is

$S_n = 2^n - 1$.

Starting with a 1-tower, one move is required, and this establishes

H1: $S_1 = 2^1 - 1$.

H2 is the statement:

H2: if $S_n = 2^n - 1$, then $S_{n+1} = 2^{n+1} - 1$.

The relationship that is known from the method of solution of the puzzle is that

$$S_{n+1} = 2S_n + 1.$$

H2 requires that from $S_n = 2^n - 1$ it should follow that $S_{n+1} = 2^{n+1} - 1$. Substituting the formula for $n+1$ in terms of n:

$$S_{n+1} = 2S_n + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1,$$

as was to be proved. Thus with *H1* and *H2* established,

C: $S_k = 2^k - 1$ for $k \geq 1$

has been proved by induction.

The 64-disk puzzle will therefore take $2^{64} - 1$ steps to solve, which is about 50,000,000,000,000 years, even if the monks move one disk a second and never make a mistake. The world appears safe enough, unless of course the monks can find some clever way to solve the puzzle faster. It is probably not in their nature to seek better methods while they work.

11.3.3 A Recursive Procedure for the Towers of Hanoi

The solution devised for the n-disk Towers of Hanoi puzzle is in terms of a solution to the $(n-1)$-disk puzzle. Thus a recursive procedure should solve the puzzle most easily, the recursion terminating when there is only one disk to move. To describe the moves, the spindles can be given character names, say **A**, **B**, and **C**. Then "move **A** to **C**" means to transfer the top disk from spindle **A** to the top of the tower on spindle **C**. In these terms the solution to the 2-disk puzzle is given by:

move **A** to **C**
move **A** to **B**
move **C** to **B**

The design for a recursive procedure to solve the puzzle is:

Design Part 3

```
PROCEDURE MoveTower(VAR Size, From, Onto, Via: CHAR);
  {Move the top Size disks from tower From
   to tower Onto via tower Via.  The tower names
   must be sensibly related and Size between 1 and 9.}
  VAR
    Size1: CHAR;
  {Include PROCEDURE PreviousDigit(VAR D1: CHAR);
   ((D1 > '0') AND (D1 <= '9') --> D1 := D1 -1) |
   (D1 = 0 --> D1 := '9')}
  PROCEDURE PrintMove(VAR Orig, Dest: CHAR);
    BEGIN {PrintMove}
      WRITELN('    ', Orig, ' -> ', Dest)
    END;  {PrintMove}
  BEGIN {MoveTower}
    IF Size > '1'
    THEN
      BEGIN
        Size1 := Size;
        PreviousDigit(Size1);
        MoveTower(Size1, From, Via, Onto);
        PrintMove(From, Onto);
        MoveTower(Size1, Via, Onto, From)
      END
    ELSE
```

```
      PrintMove(From, Onto)
  END  {MoveTower}
```

This procedure can be tested by including it in a program that assigns labels to the spindles:

```
PROGRAM HanoiPuzzle(INPUT,OUTPUT);
  VAR
    Order, S1, S2, S3: CHAR;
  {PROCEDURE MoveTower(VAR Size, From, Onto, Via: CHAR);
    {Move the top Size disks from tower From
     to tower Onto via tower Via.  The tower names
     must be sensibly related and Size between 1 and 9.}
  BEGIN {HanoiPuzzle}
    READ(Order, S1, S2, S3);
    IF (Order < '1') OR (Order > '9')
      OR (S1 = ' ') OR (S2 = ' ') OR (S3 = ' ')
      OR (S1 = S2) OR (S2 = S3) OR (S3 = S1)
    THEN
      WRITELN('Input is badly formed.')
    ELSE
      BEGIN
        WRITELN('Moves in order:');
        MoveTower(Order, S1, S2, S3);
        WRITELN('...which solves the puzzle.')
      END
  END. {HanoiPuzzle}
```

Executions

```
INPUT :2AAB
OUTPUT:Input is badly formed.
INPUT :2LMR
OUTPUT:Moves in order:
          L - R
          L - M
          R - M
        ...which solves the puzzle.
```

11.3.4 Exercises

11.3.1 Prove by mathematical induction that the sum of the natural numbers up to n,

$$0 + 1 + 2 + \ldots + n$$

is $n(n+1)/2$.

11.3.2 Work out an execution table for the 2-disk case using `HanoiPuzzle` with input <†2PQR†>.

11.3.3 Do the executions shown with `HanoiPuzzle` constitute a structural test of this program?

11.3.4 Explain why the introduction of the local variable `Size1` is or is not essential to the operation of `MoveTower`. That is, could the parameter `Size` be used instead?

11.3.5 Modify `MoveTower` to eliminate the parameters `From`, `To`, and `Via`. That is, allow the procedure to assign its own spindle names, eliminating the possibility that they will be improperly assigned.

11.3.6 Modify `MoveTower` so that it is permitted to move two disks together, so long as towers are preserved. Does this reduce the number of moves by a factor of 2? What happens to the number of moves required if larger and larger towers may be moved as a unit?

11.3.7 What changes would be needed in `MoveTower`

a) if the rule were that the tower began inverted (larger disks on top) and must be moved to another peg inverted, creating only inverted towers along the way?

b) if the tower starts off right side up, has to be moved inverted, to wind up right side up?

11.4 The Calculus of Recursive Procedures

Preview

Recursion is a powerful programming technique, but it poses difficulty in obtaining the meaning of procedure statements, for they can be defined in terms of themselves. The definition leads to a recurrence equation, which can be solved for the part function of a recursive procedure.

When there are procedure statements within the <block> of a procedure, the part-function definition of Section 9.2.5 might have to be applied within itself, to get the meaning of the inner statements as a part of obtaining the meaning for an outer procedure statement. Each procedure whose text is retrieved from the execution state might contain procedure statements, to be handled by retrieving their texts, and so on. In general there are just two possibilities for this repeated process. One: it may eventually happen that all the texts needed are retrieved, the last ones containing no procedure statements, and the meaning may then be calculated without any complication. Or, two: the procedure statements form a cycle, in which inside the text of one procedure that same procedure text is needed. This can happen directly, if a procedure calls itself recursively, or indirectly, for example, if procedure X calls Y and Y calls X. There are only these two possibilities, that the procedure statements can be eventually eliminated, or they form a cycle, because a program can contain only a finite number of procedure declarations, so any sequence of one procedure invoked within another must be finite unless one is repeated.

11.4.1 Calculating the Meaning of Recursive Procedure Calls

The definition of Section 9.2.5 is adequate for calculation in case recursion does not occur. The calculation of meaning proceeds from one procedure statement to the next, and in the end all are replaced by text whose meaning can be calculated directly. The recursive case is more interesting. The meaning cannot be calculated directly, because any attempt to calculate from the recursive text will once again require getting that text and calculating its meaning, a process with no end.

The situation is rather like the one for WHILE statements, and some of the same ideas can be used. Sometimes the recursion can be seen to terminate for a particular state, and the definition can be used directly. For example, consider the program P:

```
PROGRAM LengthOfInput(INPUT,OUTPUT);
  {Counts the number of characters modulo 10 in a
   one-line INPUT using the symbols 0 - 9.}
  VAR
    Nc, Number: CHAR;
    {Include PROCEDURE NextDigit(VAR D1: CHAR);
      D1 is replaced by its successor in the
      sequence 0123456789012...}
  PROCEDURE Count;
    BEGIN {Count}
      IF EOLN
      THEN
        Number := '0'
      ELSE
        BEGIN
          READ(Nc);
          Count;
```

```
        NextDigit(Number)
      END
    END;  {Count}
  BEGIN {LengthOfInput}
    Count;
    WRITELN(Number)
  END.  {LengthOfInput}
```

Suppose the input is <†TW†>. The state s when Count is first called is

$$s = \{\text{INPUT}\cdot\langle\dagger\dagger,\text{TW},\text{R}\rangle,\ \text{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger.\text{W}\rangle,\ \text{Nc}\cdot?,\ \text{Number}\cdot?\}$$

(omitting the procedure identifier and attached text in the interest of space). In the text $\boxed{\texttt{Count}}(s)$ the ELSE branch is taken, and when the inner call on Count occurs, the state is

$$\{\text{INPUT}\cdot\langle\dagger\text{T}\dagger,\dagger\text{W}\dagger,\text{R}\rangle,\ \text{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger.\text{W}\rangle,\ \text{Nc}\cdot\text{T},\ \text{Number}\cdot?\}.$$

Repeating this operation a third procedure statement occurs against the state

$$\{\text{INPUT}\cdot\langle\dagger\text{TW}\dagger,\dagger\dagger,\text{R}\rangle,\ \text{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger.\text{W}\rangle,\ \text{Nc}\cdot\text{W},\ \text{Number}\cdot?\}.$$

Now the EOLN IF statement within the third procedure statement takes the THEN branch, so another procedure statement does not occur, and the state that is in existence just after the third call is

$$\{\text{INPUT}\cdot\langle\dagger\text{TW}\dagger,\dagger\dagger,\text{R}\rangle,\ \text{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger.\text{W}\rangle,\ \text{Nc}\cdot\text{W},\ \text{Number}\cdot\text{0}\}.$$

This is in the midst of the second call, which can be completed now, returning to the first call the state

$$\{\text{INPUT}\cdot\langle\dagger\text{TW}\dagger,\dagger\dagger,\text{R}\rangle,\ \text{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger.\text{W}\rangle,\ \text{Nc}\cdot\text{W},\ \text{Number}\cdot\text{1}\}.$$

The original procedure statement finishes now, with the result:

$$\{\text{INPUT}\cdot\langle\dagger\text{TW}\dagger,\dagger\dagger,\text{R}\rangle,\ \text{OUTPUT}\cdot\langle\dagger\dagger,\dagger\dagger.\text{W}\rangle,\ \text{Nc}\cdot\text{W},\ \text{Number}\cdot\text{2}\},$$

and thus the result is

$$\boxed{P}(\langle\dagger\text{TW}\dagger\rangle) = \langle\dagger\text{2}\dagger\rangle.$$

Although there are potentially an infinite number of procedure statements to consider, in fact for a file containing a first line of M characters, only M+1 procedure statements will occur.

11.4.2 A Recursive Procedure Verification Rule

Just as it is not always possible in practice to expand WHILE statements until they terminate, or to see the general case so that the number of iterations can be calculated, so it is not always practical to unwind recursions. It can happen that the values of the expressions controlling the recursion are too complex for analysis and another technique is needed. For WHILE statements it was enough to unwind the loop once to obtain a recurrence equation; for recursive procedure statements such an equation can be obtained directly. There are already at least two procedure statements in the program: the original invocation, and the recursive call.

Calculating the meaning of the procedure statement in the BEGIN statement of LengthOfInput,

$$\begin{aligned}
&\boxed{\texttt{Count}} = \boxed{\texttt{BEGIN IF EOLN THEN ... END}}\\
&= \{\langle s, t\rangle : \boxed{\texttt{EOLN}}(s) = \text{TRUE},\ t = \boxed{\texttt{Number := '0'}}(s)\}\\
&\cup \{\langle s, t\rangle : \boxed{\texttt{EOLN}}(s) = \text{FALSE},\\
&\quad t = (\boxed{\texttt{READ(Nc)}} \circ \boxed{\texttt{Count}} \circ \boxed{\texttt{NextDigit(Number)}})(s)\}.
\end{aligned}$$

The function $\boxed{\texttt{Count}}$ stands explicitly on both sides of this recurrence equation. The solution to such equations in general is very difficult, but it is less hard to test a particular function to see if it is a solution. Here if we believe the comment, the meaning of Count is the function f given by the

conditional assignment

(EOLN → Number := '0') | (NOT(EOLN) → Number, Nc := m mod 10, c)

where m is the length of the future string of INPUT up to the next line marker, and c is the last character before that line marker. (The effect of Count on INPUT is ignored here.)

Substituting f for [Count] in the recurrence equation, consider two cases, an empty and nonempty future INPUT string L up to the first line marker.

Case 1: L is empty. Then for state s, the left side of the recurrence equation in f is

$f(s) = t$, where t is the same as s except that the value of Number is the character representing 0 mod 10, i.e., 0.

The right side is covered by the first set, in which [EOLN](s) = TRUE, and in that set s is paired with this same t. That is, in Case 1 f satisfies the recurrence equation.

Case 2: L is nonempty. Let the length of L be $k > 0$, and let its last character be c. Then for state s, the left side of the recurrence equation in f is

$f(s) = t$, where t is the same as s except that the value of Number is the representative for k mod 10, and the value of Nc is c.

The right side is covered by the second set, in which [EOLN](s) = FALSE, and there the value paired with s is

$t = ($ [READ(Nc)] $\circ f \circ$ [NextDigit(Number)] $)(s)$.

t can be worked out using trace tables.

CASE 2.1

Part	Condition	IN_2	Nc	Number
READ(Nc)	$L\nabla/ \neq \dagger\dagger$	$L\ \nabla\ /$	$\Theta(L\nabla/)$	
f	$\Theta(\Lambda(L\nabla/)) = /$	$\Lambda(L\nabla/)$	c	0
NextDigit...				1

Condition: $(\Theta(L\nabla/) \neq \dagger\dagger)$ AND $(\Theta(\Lambda(L\nabla/)) = /)$

Assignment: Number, Nc := '1', c

The condition is only true if L is a 1-string, because

$\Lambda(L\nabla/) = \nabla/$

so

$\Theta(\Lambda(L\nabla/)) = /$

evaluates to TRUE. Also, since L is a 1-string, $\Theta(L\ \nabla\ /) = c$ is the only character in L (before the line marker). Thus, ignoring the effects on INPUT, the conditional assignment is

($length(L)$ = 1 → Number, Nc := '1', c)

CASE 2.2

Part	Condition	IN_2	Nc	Number
READ(Nc)	$(L\nabla/) \neq \dagger\dagger$	$L\nabla/$	$\Theta(L\nabla/)$	
f	$(\Theta(\Lambda(L\nabla/))) \neq /$	$\Lambda(L\nabla/)$	c	$length(\Lambda(L))$ mod 10
NextDigit...				$(length(\Lambda(L))$ mod 10 + 1) mod 10

(NextDigit increments Number except for value 9, which becomes 0. This behavior is addition modulo 10.)

Condition: $((L\nabla/) \neq \dagger\dagger)$ AND $(\Theta(\Lambda(L)) \neq /)$

Assignment: Number, Nc := $((length(\Lambda(L)) \bmod 10) + 1) \bmod 10, c$

The condition can be simplified by observing that L must be at least a 2-string because $\Lambda(L)$ does not start with a line marker. Thus, ignoring the effects on INPUT, the conditional assignment is

$(length(L) > 1 \rightarrow$ Number, Nc := $((length(\Lambda(L)) \bmod 10)+1) \bmod 10, c)$

Note that

$$length(\Lambda(L)) \bmod 10 + 1 = length(L) \bmod 10$$

whenever $length(L) \geq 1$, and that applying a second mod function has no effect. Cases 2.1 and 2.2 can be combined to yield:

$(length(L) >= 1 \rightarrow$ Number, Nc := $length(L) \bmod 10, c)$

This is exactly what was obtained for the left side, so f satisfies the recurrence equation in case 2. It has been shown that the function f derived from a comment in LengthOfInput satisfies the recurrence equation for $\boxed{\texttt{Count}}$ in that program.

Care is required, as it was with the WHILE statement, however, since the recurrence equation is only a necessary condition. It could happen that a function satisfies the equation, but is not at all the meaning of the procedure from which the equation was derived. The crucial example is the procedure

```
PROCEDURE Forever;
  BEGIN
    Forever
  END;
```

whose recurrence equation is

$$\boxed{\texttt{Forever}} = \boxed{\texttt{Forever}}$$

and any function whatsoever is a solution. For this procedure the meaning should be the empty function, since the recursion never terminates. By analogy with the WHILE statement case, this suggests a verification rule: the part function for a procedure statement involving a recursive call is a solution to the recurrence equation whose domain agrees with the one for the procedure.

Let P be a procedure statement for a procedure without parameters and with one recursive call, having the recurrence equation $\boxed{P} = M(\boxed{P})$. $M(\boxed{P})$ represents the result of calculating the part function of the procedure <block>, in which P occurs just once. This calculation must be carried out until $\boxed{P}$ occurs explicitly, as in the example above. A function $f = \boxed{P}$ if and only if:

1) domain(f) = domain($\boxed{P}$), and

2) f is a solution to the recurrence equation.

11.4.3 Examples of Recursive Procedure Verification

To illustrate the rule, it has been shown above that the function f:

(EOLN $\rightarrow$ Number := '0') | (NOT(EOLN) $\rightarrow$ Number, Nc := $m \bmod 10, c$)

satisfies the recurrence equation in $\boxed{\texttt{Count}}$:

$$\boxed{\texttt{Count}} = \{<s, t>: \boxed{\texttt{EOLN}}(s) = \text{TRUE}, t = \boxed{\texttt{Number := '0'}}(s)\}$$
$$\cup \{<s, t>: \boxed{\texttt{EOLN}}(s) = \text{FALSE},$$
$$t = (\boxed{\texttt{READ(Nc)}} \circ \boxed{\texttt{Count}} \circ \boxed{\texttt{NextDigit(Number)}})(s)\}.$$

This is condition 2) of the verification rule. Therefore, to complete a proof that $f = \boxed{\texttt{Count}}$ requires only checking condition 1). The domains of f and $\boxed{\texttt{Count}}$ are evidently the same, since both have the same conditions on EOLN as part of their definitions.

As another example of a recursive procedure verification, consider a procedure Rev that is intended to reverse a line of INPUT to OUTPUT:

```
PROCEDURE Rev;
  VAR
    Ch: CHAR;
  BEGIN {Rev}
    IF NOT(EOLN)
    THEN
      BEGIN
        READ(Ch);
        Rev;
        WRITE(Ch)
      END
  END {Rev}
```

The recurrence equation for a procedure statement using Rev is:

$$\boxed{\texttt{Rev}} = \boxed{\texttt{VAR Ch: CHAR}} \circ \boxed{\texttt{BEGIN IF ... END}} \circ \boxed{\texttt{VAR Ch: CHAR}}^{T}$$

where

$$\boxed{\texttt{BEGIN IF ... END}}$$
$$= \{<s, s>: \boxed{\texttt{EOLN}}(s) = \text{TRUE}\} \cup$$
$$\{<s, t>: \boxed{\texttt{EOLN}}(s) = \text{FALSE}, t = (\boxed{\texttt{READ(Ch)}} \circ \boxed{\texttt{Rev}} \circ \boxed{\texttt{WRITE(Ch)}})(s)\}.$$

Substituting in the right side of the recurrence equation, the first set makes no changes in the state, so the $\boxed{\texttt{VAR Ch: CHAR}}$ and its transpose cancel, giving the equation as

$$\boxed{\texttt{Rev}} = \{<s, s>: \boxed{\texttt{EOLN}}(s) = \text{TRUE}\} \cup$$
$$\{<s, t>: \boxed{\texttt{NOT(EOLN)}}(s), t = (\boxed{\texttt{VAR Ch: CHAR}} \circ$$
$$\boxed{\texttt{READ(Ch)}} \circ \boxed{\texttt{Rev}} \circ \boxed{\texttt{WRITE(Ch)}} \circ \boxed{\texttt{VAR Ch: CHAR}}^{T})(s)\}.$$

The intended function for Rev is

$$f = \{<s, s>: \boxed{\texttt{EOLN}}(s)\} \cup$$
$$\{<s, t>: \boxed{\texttt{NOT(EOLN)}}(s), \boxed{\texttt{INPUT}}(s) = <x,((y \nabla c) \nabla /) \,\&\, z,\mathbf{R}>,$$
$$\boxed{\texttt{INPUT}}(t) = <x \,\&\, (y \nabla c),(\nabla /) \,\&\, z,\mathbf{R}>,$$
$$\boxed{\texttt{OUTPUT}}(t)_1 = (\boxed{\texttt{OUTPUT}}(s)_1 \nabla c) \,\&\, \mathit{reverse}(y)\},$$

where x and z are strings, y is a string containing no line markers /, c is a character, and $\mathit{reverse}(y)$ is the reversal of string y.

It remains to prove that $f = \boxed{\texttt{Rev}}$ using the verification rule.

Condition 1 requires that the domains of these function agree. Both domains are evidently all states for which $\boxed{\texttt{EOLN}}$ is defined. (The condition on INPUT in f is not restrictive, but merely gives the general form when EOLN is FALSE.)

Condition 2 requires that f satisfy the recurrence equation. Substituting, this is:

$$f = \{<s, s>: \boxed{\texttt{EOLN}}(s) = \text{TRUE}\} \cup$$
$$\{<s, t>: \boxed{\texttt{NOT(EOLN)}}(s), t = (\boxed{\texttt{VAR Ch: CHAR}} \circ$$
$$\boxed{\texttt{READ(Ch)}} \circ f \circ \boxed{\texttt{WRITE(Ch)}} \circ \boxed{\texttt{VAR Ch: CHAR}}^{T})(s)\}.$$

There are two cases, depending on whether INPUT is at end of line or not.

Case 1: INPUT is at end of line in state s.

The left side of the recurrence equation in f is $f(s) = s$, because in the definition of f, the first set applies. On the right side of the equation the first set also applies, with the result s. That is, in case 1 f satisfies the equation.

Case 2: INPUT is not at end of line in state s.

The left and right sides of the equation can be worked out in trace tables, with two conditions for the right side depending on whether INPUT has more than one character:

Left side of recurrence equation

Part	Condition	INPUT	OUTPUT
	NOT (EOLN)	$\langle x,((y \nabla c) \nabla /) \,\&\, z,\mathbf{R}\rangle$	$\langle u,\dagger\dagger,\mathbf{W}\rangle$
f		$\langle x \,\&\, (y \nabla c),(\nabla /) \,\&\, z,\mathbf{R}\rangle$	$\langle (u \nabla c) \,\&\, reverse(y),\dagger\dagger,\mathbf{W}\rangle$

Right side of recurrence equation (one-character input line)

Part	Condition	INPUT	OUTPUT	Ch
	NOT (EOLN)	$\langle x,((\nabla c) \nabla /) \,\&\, z,\mathbf{R}\rangle$	$\langle u,\dagger\dagger,\mathbf{W}\rangle$	
VAR Ch: CHAR				?
READ (Ch)		$\langle x \nabla c,(\nabla /) \,\&\, z,\mathbf{R}\rangle$		c
	EOLN			
f		$\langle x \nabla c,(\nabla /) \,\&\, z,\mathbf{R}\rangle$		
WRITE (Ch)			$\langle u \nabla c,\dagger\dagger,\mathbf{W}\rangle$	

Right side of recurrence equation (y not empty)

Part	Condition	INPUT	OUTPUT	Ch
	NOT (EOLN)	$\langle x,((y \nabla c) \nabla /) \,\&\, z,\mathbf{R}\rangle$	$\langle u,\dagger\dagger,\mathbf{W}\rangle$	
VAR Ch: CHAR				?
READ (Ch)		$\langle x \nabla(\Theta\, y),$ $(((\Lambda\, y) \nabla c) \nabla /) \,\&\, z,\mathbf{R}\rangle$		$\Theta\, y$
	NOT (EOLN)			
f		$\langle x \,\&\, (y \nabla c),$ $(\nabla /) \,\&\, z,\mathbf{R}\rangle$	$\langle (u \nabla c) \,\&\, reverse(\Lambda\, y),$ $\dagger\dagger,\mathbf{W}\rangle$	
WRITE (Ch)			$\langle (u \nabla c) \,\&$ $(reverse(\Lambda\, y) \nabla (\Theta\, y)),$ $\dagger\dagger,\mathbf{W}\rangle$	

In these tables, x, u and z are strings, y is a string containing no line markers /, and c is a character.

When there is a single character in the input line, y is empty, and the values on the left and right side reduce to the same thing. When there is more than one character, the values for INPUT are the same, and the right side OUTPUT is

$$\langle (u \nabla c) \,\&\, (reverse(\Lambda\, y) \nabla (\Theta\, y)),\dagger\dagger,\mathbf{W}\rangle$$

but

$$reverse(\Lambda\, y) \nabla (\Theta\, y) = reverse(y)$$

so the two sides are the same in this case, too. Thus condition 2 of the verification theorem is satisfied, and the intended function f is the same as the actual part function Rev.

11.4.4 Proof of the Recursive Procedure Verification Rule

The verification rule applies to a procedure statement P for a procedure without parameters and with one recursive call, having the recurrence equation $\boxed{P} = M(\boxed{P})$. A function $f = \boxed{P}$ if and only if:

1) domain(f) = domain($\boxed{P}$), and

2) f is a solution to the recurrence equation.

The only-if direction of the proof is trivial, since $\boxed{P}$ satisfies both conditions, and hence so does $f = \boxed{P}$.

For the other direction, assume conditions 1) and 2). Consider any $s \in$ domain($\boxed{P}$). When the right-hand side of the recurrence equation is applied to s, by the time $\boxed{P}$ itself is applied, the state may have been altered (by other statements in $M(\boxed{P})$) to s'. In evaluating $\boxed{P}(s')$ again use the right side of the recurrence equation, and continue this process of evaluating the parts of $M(\boxed{P})$ and substituting the right side of the recurrence equation whenever $\boxed{P}$ itself is to be applied to the modified state. The process cannot continue indefinitely, because $\boxed{P}$ is defined at s; the recursion must terminate. That is, after (say) k substitutions in the right side of the recurrence equation, the part of $M(\boxed{P})$ involving $\boxed{P}$ itself will not appear, and the calculation of the resulting state can be completed. That is, combining these substitutions into a k-fold application of the modification M,

$$\boxed{P}(s) = (M^k(\boxed{P}))(s) = t,$$

where the state t involves no application of $\boxed{P}$.

Now consider writing the recurrence equation in terms of f instead of $\boxed{P}$. Since f solves the equation by condition 2), making k substitutions gives

$$f(s) = (M^k(f))(s) = t,$$

the same state obtained when using the equation for $\boxed{P}$, because this state in no way depends on the application of either f or $\boxed{P}$, only on the other statements from $M(\boxed{P})$. Thus $\boxed{P}(s) = f(s)$, and since s was an arbitrary point in their common domain, $\boxed{P} = f$ as was to be proved.

Another way to characterize the function that captures the meaning of a recursive procedure call is that it is the *least-defined* solution to the procedure's recurrence equation. That is, F is the function satisfying the equation which is undefined as often as possible. (No other function satisfying the equation is a subset of F.) To see why this statement is equivalent, notice that if F has the same domain as $\boxed{P}$, F satisfies the verification rule, and hence $F = \boxed{P}$. Suppose then that domain(F) were strictly smaller than domain($\boxed{P}$), that there were a point x for which $\boxed{P}$ is defined but F is not. Substitute in the recurrence equation for $\boxed{P}$ until it disappears for argument x as above,

$$\boxed{P}(x) = y,$$

where y does not involve an application of $\boxed{P}$. But since F satisfies the same equation,

$$F(x) = y$$

as well, and thus F *is* defined at x. The same argument applies to any point in domain($\boxed{P}$), so the least solution F has this domain.

In a practical verification, the original form of the rule is more useful than the least-defined form, because it is easier to identify the domain of the actual procedure statement part function than to show that a trial solution to the recurrence equation is the least-defined one.

The meaning of a procedure statement for a recursive procedure must therefore be obtained by finding a trial function, then verifying that it is the meaning by use of the above rule. The situation is similar to that for WHILE statements presented in Section 8.4.2. However, the WHILE statement verification rule applied to the general case, and the result above is restricted to a single recursion where there are no parameters. The technical details of the general case for recursion are considerable, since it can involve several simultaneous recurrence equations for a multiple recursion, and different parameters to the same procedure if it occurs several times in these equations.

11.4.5 Exercises

11.4.1 Calculate Never, for

```
PROCEDURE Never;
  BEGIN
    IF '0' = '1'
    THEN
      Never
  END
```

in two ways:

a) by explicitly unwinding the recursion, and

b) using the verification rule for recursive procedures.

11.4.2 For the recurrence equation of Count in LengthOfInput,

a) Show that the empty function is *not* a solution.

b) Show that the identity function is *not* a solution.

11.4.3 Rewrite the procedure RSort of Section 11.1.1 so that it has no parameters, but instead uses a global variable.

a) Write the recurrence equation for the procedure of a) and expand the calculation until RSort appears explicitly on the right side.

b) Give a candidate function for the meaning of RSort.

c) Using the verification rule for recursive procedures, prove that the candidate of b) is in fact RSort

11.4.4 Repeat the previous exercise for the procedure Palindrome of Section 11.1.2.

11.5 Chapter Summary

Recursion is a programming technique that can often replace iteration to solve a problem in a simple and elegant way. Recursive solutions are appropriate when the problem can be broken into solving a simpler version of itself, and then modifying that simple solution to get the solution to the actual problem.

CF Pascal procedures may invoke themselves, and a recursive procedure is the mechanism to implement a recursive problem solution. An important aspect of programmed recursion is that each new invocation of a recursive procedure includes local variables that are distinct from those of previous invocations. This feature is essential if the many procedures, active at the same time, are to keep straight their separate tasks.

Sorting and reversing using recursion are good examples of the technique. The Towers of Hanoi puzzle has a particularly elegant recursive solution.

Mathematical induction can be useful in proving properties of recursive problems.

A verification rule for simple recursive procedures is very similar to the WHILE statement verification rule given in Chapter 8.

CHAPTER 12

SYSTEMATIC PROGRAM DESIGN AND CORRECTNESS

Chapter Preview

The techniques of program design, and verification that a design is correct, can be formalized. The specification for a program, a statement of the problem and what the program should do, plays the central role in this formalization. Careful mathematical formalism plays a role even in large designs where full details are impractical to obtain.

12.1 Program Specifications

Preview

A program specification describes a problem and its solution as a relation between input and output strings.

New ideas: specification.

Any program has a purpose. Usually, that purpose is to solve some problem. Data for the problem will be given in the form of an input character string, and the solution will be described by giving the output string the execution of the program should produce. These input and output strings may contain words and numbers which are meaningful to the problem solver but which are read and written as strings of characters, one at a time, by the program. A *program specification* for a problem defines the set of input strings which represent meaningful data for the problem and for each input string defines the set of (one or more) output strings which are acceptable solutions for that problem statement. Thus a specification is a relation consisting of all pairs of acceptable instances of input data and output data for a problem. The domain of this relation is all acceptable inputs. For character strings not in the domain of this relation, there is no required program behavior.

Here are three examples:

A specification for sorting character strings. The input is any character string. For each input string only one output string is acceptable, namely the character string with exactly the same elements as the input but in sorted order. The specification will be a function in this case because every input defines a unique output.

A specification to evaluate an arithmetic expression written with natural numbers. Most character strings are not allowed as inputs, only those that represent the arithmetic expressions with natural-number operands. (These might be precisely described using BNF, much as the <expression>s of CF Pascal are described.) The output string must be a representation of a single natural number, which is the value of the evaluated input expression. Unless there is latitude given about how the result is to appear in the string, its base, etc., this specification is a function.

A specification to evaluate a character set expression. The input will be only those strings representing set expressions, with the set elements listed explicitly. The output is to represent the result of evaluating the expression. The output may not be unique because the elements of sets can be listed in different orders, so a set with two or more elements can be represented by more than one character string. In this case the specification is a relation that isn't a function.

These three examples illustrate that program specifications may be functions or relations and their domains may be restricted.

Although some specifications seem to make more sense than others in a given problem context, there are no ironclad rules about specifications. For example, in a specification concerning the evaluation of arithmetic expressions, it may be important to recognize the strings that are not arithmetic expressions and produce error messages. The very set of error messages produced for various deviations from correct arithmetic syntax could be part of the specification. Thus, two specifications which both required the same arithmetic but differed in their treatment of incorrect syntax would be different specifications. A specification might even be in the form of "if the arithmetic syntax is incorrect put out an error message." Then the programmer is free to create any error message(s). The specification is a relation in which every correctly formed arithmetic expression is associated with a unique output but an incorrectly formed arithmetic expression corresponds to any output which is recognizable as an error message. (Evidently it would be unwise to number the errors and have messages like **125** that could be confused with expression values.)

For problems involving complex text inputs or outputs, BNF is useful for describing the domain and range of a specification. However, BNF usually cannot make the correspondence between inputs and outputs that is an essential part of specification. Whenever specifications include informal, English-language parts, there is danger of ambiguity. For example, a specification for sorting words in text could define the domain and range strings using BNF. But the additional statement that the output is to be the words of the input in sorted order raises several questions: Should repeated words in the input string be repeated in the output or should duplicate words be eliminated? Is the number of blanks separating the output words important? Is the sort to be in ascending or descending order?

It is not always desirable to eliminate ambiguities from program specifications. What is needed are program specifications that are well conceived with respect to the problem being solved. If the output is to be used by another program, then exact format may allow another program to be written more simply. If the output is to be used by a human, then a program may be simpler if given more freedom to provide output in the most convenient of various possible forms.

12.1.1 A Specification for RemoveExtraBlanks

The problem solved by `RemoveExtraBlanks` in Section 10.1.3 was described in English, and BNF for its input and output forms was developed as part of the design. Attaching the subscript S for specification, it was:

1. <FileIn>$_S$ ::= <WBlist> <blank> ¦ <blanks> <WBlist> <blank>
2. <WBlist> ::= <empty string> ¦ <word> <blanks> <WBlist>
3. <word> ::= <nonblank> ¦ <word> <nonblank>
4. <blanks> ::= <blank> ¦ <blanks> <blank>
5. <FileOut>$_S$ ::= <Wlist> ¦ <empty string>
6. <Wlist> ::= <word> ¦ <word> <blank> <Wlist>

CR. The <word>s of <Wlist> must be the same as those of <WBlist>, in the same order.

The BNF description is very precise and formal, but the context rule could be improved. The appropriate level of formality depends upon the knowledge and understanding of the specification's reader, and its importance to the people for whom the problem is being solved. One way to make the description of the words and their order more formal is to define a function that maps the words of a string to a list of words, preserving the order. Call this function *words*. For example:

words(† **This is time is** †) = <†**This**†, †**is**†, †**time**†, †**is**†>.

When we want to refer to all the possible strings that are defined by a BNF rule, we use the rule's name, for example we talk about a <block> in CF Pascal. Then the input for the problem of removing extra blanks is a <FileIn>$_S$ and the output is a <FileOut>$_S$, related by the context rule

CR. *words*(<FileIn>$_S$) = *words*(<FileOut>$_S$),

with the meaning that the strings on which *words* operates are those described by any <FileIn>$_S$

string and corresponding <FileOut>$_S$ string, respectively. This specification can be written as:

$$r = \{<<\text{FileIn}>_S, <\text{FileOut}>_S>: \mathit{words}(<\text{FileIn}>_S) = \mathit{words}(<\text{FileOut}>_S)\}.$$

A program specification is a relation describing the inputs and acceptable outputs for a problem. It may be difficult to write down precisely, but it is a mathematical relation nevertheless. The specification of a large system may occupy hundreds of pages of English, augmented by engineering drawings, mathematics, or descriptions drawn from the field of the problem, but it is still just a relation.

12.1.2 Exercises

12.1.1 Write a program specification for addition of several natural numbers, where the problem is presented in infix notation. Describe the syntax of input and output strings using BNF, and define a function *number* that maps strings representing numbers to the numbers. Use these elements in the specification.

12.1.2 Write a program specification for printing all four-letter words found in a character string, in the same order as found.

12.1.3 Write a program specification for the problem of Sara Revere's Letter from Section 2.3.3.

12.1.4 Write a program specification for the problem of the Towers of Hanoi from Section 11.3.1.

12.1.5 Given a program P, we have learned in the previous chapters how to calculate $\boxed{P}$. Comment on the following statement: Since P has been written to solve some problem, and since $\boxed{P}$ is a function, $\boxed{P}$ should be used for the specification of that problem.

12.1.6 Criticize the following as a specification of sorting characters from one file into another, where duplicate characters of the input are to occur just once in the output:

> The output file is to be a sequence of characters such that each character in the sequence comes strictly after the one before it in alphabetical order.

Give a more precise specification for this problem.

12.1.7 Some people believe that specifications should never be relations that are not functions, because a function better pins down exactly what is to be done. Give a brief argument *against* this position.

12.1.8 Specifications often have "don't care" parts. For some inputs, the output is of no interest, so the author of the specification, if questioned about what should be done for those inputs, answers "I don't care." But how should this be reflected in the specification relation? Two possibilities suggest themselves: (1) the relation should include all possible values for output paired with that input; or, (2) the relation should include no pairs with that input. Explain which you think is a better definition to take.

12.1.9 Describe a problem whose specification relation might be said to be a "specification for the programming language CF Pascal." What are the input values, and what are the outputs? This specification should certainly include the BNF description of CF Pascal. What is missing if only the BNF were used?

12.2 Program Correctness

Preview

Formally, correctness is a property of the program function and the specification relation, a property that can be proved using set theory. Effort should be split wisely between program design and program proving. The usual mistake is a too ambitious design that gets out of intellectual control.

The specification relation describes what a program should do; it expresses intentions that are yet to be realized. It exists apart from any program that might be written to realize it. In fact, the specification is the starting point for programming. It is sound practice to have the specification well in mind before designing a program or developing a strategy for its design.

On the other hand, once a program exists, its program function defines what the program does do, without regard for intentions the programmer may have had. Furthermore, using the program calculus the program function can be calculated directly from the text itself. Neither the specification relation nor the program function describes how the program accomplishes what it does. But the program function calculation depends on full details of the program's inner workings.

A central question of programming and certainly a central question for a programmer trying to meet a specification, can be simply stated in these terms:

Given a specification and a program does the program fulfill that specification?

Informally, if the answer to the question is "yes," then the program is said to be correct with respect to the specification. More formally, given a program specification relation r and a program P, we say that P is correct with respect to r if, for every member x of the domain of r (an instance of input data), P produces some member of the range of r which corresponds to x. That is, for each input x, P produces result y such that $\langle x, y\rangle \in r$. What P does to input data not in the domain of r is irrelevant since r defines all behavior important to the problem solver. The definition is reduced to a purely set-theoretic idea in the following theorem:

Correctness Theorem. Program P is correct with respect to specification relation r if and only if

$$\text{domain}(r \cap \boxed{P}) = \text{domain}(r).$$

Proof: The expression $(r \cap \boxed{P})$ identifies all acceptable pairs of r computed by P. Therefore $\text{domain}(r \cap \boxed{P})$ identifies the set of input data for which P produces acceptable output data. Since $\text{domain}(r)$ is the set of input data for which r specifies acceptable output data, the condition

$$\text{domain}(r \cap \boxed{P}) = \text{domain}(r)$$

ensures that P produces acceptable output data for every instance of input data defined by r. This completes the proof of the theorem.

P may execute successfully for input data not identified by r, but such pairs of $\boxed{P}$ are screened out of $(r \cap \boxed{P})$ by r. If P produces an unacceptable instance of output data, no member of r with that input data can be in $(r \cap \boxed{P})$, and therefore $\text{domain}(r \cap \boxed{P})$ cannot equal $\text{domain}(r)$. For example, if

$$r = \{\langle \mathbf{A}, \dagger\mathbf{A}\dagger\rangle, \langle \mathbf{B}, \dagger\mathbf{A}\dagger\rangle, \langle \mathbf{A}, \dagger\mathbf{B}\dagger\rangle, \langle \mathbf{B}, \dagger\mathbf{B}\dagger\rangle\}$$

and

$$\boxed{P} = \{\langle x, \nabla x\rangle : x \text{ is a character}\},$$

then

$$r \cap \boxed{P} = \{\langle \mathbf{A}, \dagger\mathbf{A}\dagger\rangle, \langle \mathbf{B}, \dagger\mathbf{B}\dagger\rangle\}$$

and $\text{domain}\,(r \cap \boxed{P}) = \text{domain}\,(r) = \{\mathbf{B}, \mathbf{A}\}$.

In case the program specification is a function, the condition for program correctness can be simplified as follows:

Corollary: Program P is correct with respect to specification function f if and only if

$$f \subseteq \boxed{P}.$$

Proof: The expression $f \cap \boxed{P}$ identifies all acceptable pairs of f computed by P, which must be f, itself. That is, P is correct with respect to f if and only if

$$f \cap \boxed{P} = f.$$

Therefore $f \subseteq \boxed{P}$, which completes the proof of the corollary.

Consider the correctness of `RemoveExtraBlanks` whose specification r was given above. Analysis of the program in Section 10.1.5 showed that the input and output files actually processed are described by the BNF rules (using P for the program):

<FileIn>$_P$ ::= <File1> ¦ <blanks> <File1>
<File1> ::= <R> ¦ <blanks>
<R> ::= <word> <blanks> <R> ¦ <word> <blanks>
<FileOut>$_P$ ::= <File2>
<File2> ::= <S> ¦ <empty string>
<S> ::= <word> <blank> <S> ¦ <word>

and that the <word>s moved from input to output are the same. That is,

$$[\![P]\!] = \{\langle \langle \text{FileIn}\rangle_P, \langle \text{FileOut}\rangle_P\rangle : \mathit{words}(\langle \text{FileIn}\rangle_P) = \mathit{words}(\langle \text{FileOut}\rangle_P)\}.$$

<FileOut>$_P$ and <FileOut>$_S$ are evidently the same, since <S> and <Wlist> are the same. <FileIn>$_P$ and <FileIn>$_S$ are also the same, because <R> describes the same strings as <WBlist> <blanks>. Thus the specification and program function are the same,

$$[\![P]\!] = r,$$

and `RemoveExtraBlanks` is correct.

12.2.1 Mathematical Formality and Proof

Mathematical formality is a limited resource, to be used wisely just as is computer time for testing. It is pointless to be extremely formal, then run out of time with a program only ten percent proved. It is better to adjust the level of formality to the circumstance. A simple data-transfer operation may be verified by direct inspection, while a complex priority-queue sorting operation may require a full-scale conditional trace table treatment. The budget between program design and program proving should also be in balance. It may be better to design a less capable program and do more formal proving, than to attempt an ambitious design that gets out of intellectual control.

Long division is mathematics, and so is proving that a program is correct with respect to a specification. When we do long division, we can make mistakes—for example, we might set down 9 x 6 = 63 by mistake, an error of execution. But a more serious mistake is to put the digits 6 and 3 in the wrong columns of a long division layout, a mistake in methodology. We can catch each other's mistakes of execution more reliably than mistakes in methodology. For example, an onlooker might imagine a methodological mistake to be another legitimate way of solving the problem. At the point where the onlooker ceases to understand the method, all checking stops. Similarly, in showing a program correct, we might decide that two conditional assignments are the same, but be mistaken. But a more serious mistake is to compare two functions when we should be composing them and comparing the result to one (as in the WHILE verification rule)—then the method has gone wrong and the onlooker ceases to understand. Once precise communication breaks down, anything can be "proved" in the world of vague ideas.

Doing mathematics is a human activity subject to human fallibility. But it is a human activity with centuries of experimental validation and represents the best knowledge of humanity about logical reasoning processes. Mathematics permits a team activity with minimum emotional attachment and defense of mistakes.

12.2.2 Mathematical Proofs

A mathematical proof is an attempt at human persuasion, which can be regarded as a repeatable experiment, much like those of the chemistry laboratory. The difference is, instead of physical objects, the experimental subject is a human being. The experiment consists of a sequence of assertions, of the form, "from so-and-so it follows that such-and-such," and if the subject agrees, the experiment continues. If successful, the experiment leads to agreement on the entire sequence. If the subject does not agree with some assertion, the experiment fails. The experiment may be successful with an incorrect proof, or it may fail with a correct proof, because of human fallibility.

Mathematical notation plays no direct role in a proof except to facilitate reasoning and calculation by previously agreed-upon procedures. Indeed, mathematical logic only formalizes previously accepted rules for "agreeing it is obvious," and computers can be used to follow such rules when they are made precise enough. As a human activity, the design of a mathematical proof shows many similarities to the design of a program. A complex proof, as a complex program, can only be understood as a hierarchy of subproofs. But whereas programs are executed by machines, proofs are executed by human beings, and not only human fallibility, but human attention span and motivation need to be considered. It does no good to produce a perfect 1000-assertion proof if the subject loses interest and stops listening after 100 assertions. It does no good to produce a perfect 10-step proof if the steps are too large for the subject to accept. And just as the level of formality should differ for different uses of the same program, a subject will tolerate more proof steps for the same proof on one occasion than on another, depending on its importance at the moment. So the strategy of proof should account for the occasion. There is no right level for every occasion, and no ultimate level of proof.

In one circumstance the best strategy for proving a program correct might be to make the assertion: "see for yourself, this program is obviously correct with respect to its specification." It is a legitimate proof, and for a simple program no more effort may be warranted. However, the total effort required to understand and agree on the correctness of a program is usually decreased by a proof of some moderate length. In fact, a proof of correctness serves as the best documentation for a program. So an outline of the main design's statement functions serves as the minimal proof, i.e., "these statement functions obviously combine to satisfy the specification."

All this is to say that proving program correctness explicitly is not a luxury of logicians, but a necessity for prudent programmers. No proof at all is really the one-assertion proof "it is obvious." Any documentation is a kind of proof. So the question is not whether proofs should be made, but only what form they should take.

12.2.3 Exercises

12.2.1 Explain why the following conditions are not satisfactory as definitions for program correctness:

a) $r \cap \boxed{P} = r$

b) $\text{range}(r \cap \boxed{P}) = \text{range}(r)$

c) $r \cap \boxed{P} = \boxed{P}$

d) $\text{domain}(r \cap \boxed{P}) = \text{domain}(\boxed{P})$

12.2.2 Give a formal definition of "program P is correct with respect to specification r at input x." Show that your definition, prefaced with "for every x," is equivalent to the one given in this section.

12.2.3 When a programmer writing program P "omits a case" in solving a problem, because he or she did not think of the possibility of some input value x, what does this mean about pairs in $\boxed{P}$ with input value x? Suppose that in fact the specification relation for P does not include any such pairs with input x. If the program has no other mistakes, explain why it is then correct, using the definition of this section.

12.2.4 A particular specification includes some "don't-care" inputs, pairing them with all possible output values. Explain why it is still possible to write a program that is not correct for one of the don't-care inputs.

12.3 Program Design

Preview

Program parts may be given specifications, which may be conditional assignments. Then a program part can be proved correct. The design of a program using stepwise composition is a process of choosing a function, then refining it into simpler functions that compose it. These functions are the specifications to which program parts must be written.

The program calculus has been developed for CF Pascal programs in complete detail, but to handle large program designs its level of formality is too high. A way is needed to scale up to large programs without losing the mathematical discipline of designing to precise functions. Because the calculus works with sets and functions, these can be described less formally, yet still maintain the design discipline.

12.3.1 Correctness of Program Parts

Program correctness can be carried over to correctness of program parts. Program meanings map input strings to output strings, but program-part meanings map execution states to execution states. A specification for a program part will therefore be a relation whose domain and range are both sets of execution states. A program-part meaning can be more precise and constrained than that of a program which involves an outside problem statement and solution for a user. Nevertheless, there may be cases where a specification is better stated with the freedom of a relation than with a function.

Good techniques for the specification of program parts are similar to those for the specification for programs. If strings are contained in the execution states then BNF can be helpful in describing them. The construction of auxiliary functions or relations related to the problem being solved will often be useful in describing the specification beyond the BNF.

12.3.2 Conditional Assignments as Specifications

The conditional assignment is powerful enough to express the effect of any CF Pascal program part. Conditional assignments serve two simultaneous purposes in designing CF Pascal programs:

1. As the components of a high-level design, and
2. As intended meanings for program parts that are to implement them.

In decomposing a problem into a design at any level, the conditional assignments serve as the components of the design, and allow calculation of its effect. Then these same assignments, considered as intended meanings, can be used to verify the parts that implement them.

Conditional assignments can be described in any convenient mixture of CF Pascal, mathematics, and English. For example, the procedure

```
PROCEDURE Sort2(VAR Ch1, Ch2: CHAR);
  VAR
    Temp: CHAR;
  BEGIN
    IF Ch1 > Ch2
    THEN
      BEGIN
        Temp := Ch1;
        Ch1 := Ch2;
        Ch2 := Temp
      END
  END
```

can be expressed as the design

```
PROCEDURE Sort2(VAR Ch1, Ch2: CHAR);
  VAR
    Temp: CHAR;
  BEGIN
    {(Ch1 > Ch2 -->
      Ch1,Ch2,Temp := Ch2,Ch1,Ch1) |
     (Ch1 <= Ch2 -->   )}
  END
```

or

```
PROCEDURE Sort2(VAR Ch1, Ch2: CHAR);
  VAR
    Temp: CHAR;
  BEGIN
    IF Ch1 > Ch2
    THEN
      {Ch1,Ch2,Temp := Ch2,Ch1,Ch1}
  END
```

or even

```
PROCEDURE Sort2(VAR Ch1, Ch2: CHAR);
  BEGIN
    IF {Ch1 <> min(Ch1,Ch2)}
    THEN
      {Ch1,Ch2 := Ch2,Ch1}
  END
```

12.3.3 Program Design by Function Refinement

Since any CF Pascal program has a program function, the task of program design can be treated in two steps:

1. Find a function that satisfies the program specification.
2. Design a program which has this function as its meaning.

In some cases the function found may be influenced by the program design itself. For example, if a program is required to produce the divisors of any input number, any function of the form

{<input number, list of divisors>}

will do, where the format, in particular the list order, is not material. But there may be a simple way to determine the divisors in numerical order, and that particular function will have the simplest program design. In other cases, the choice of the function may not be so easily seen, but may only become clear after considerable thinking and experimentation with design ideas.

In any case, we concentrate on Step 2 of the design task, noting that the BEGIN statement of a CF Pascal program (or procedure) which determines its meaning is composed of composite statements of just three kinds:

1. BEGIN statements
2. IF statements
3. WHILE statements

Each of these is in turn composed of composite statements of the same kinds.

The program calculus allows us to redescribe the process of stepwise composition as a process of function refinement. At each step, an intended function is given, and a composite statement type is selected. Then a CF Pascal design of that statement is created, defining appropriate functions for the constituent statements. If a procedure statement is used as a constituent, it will be ultimately treated as a BEGIN statement. Any other statements are composite or simple statements. The refinement stops with the simple statements (whose part functions are correspondingly simple), while

the refinement continues for any composite statement introduced. At first glance, the shift from stepwise composition to function refinement may seem to complicate matters. But its practice, while frustrating and seemingly unnatural at first, soon leads to a new capability for abstraction and self-discipline in program design.

Program design by function refinement requires the designer to think about programs in "wholes," that is, about the total program behavior, for all possible input data, and all possible cases of execution.

12.3.4 Exercises

12.3.1 Are the mappings that go with conditional assignments ever relations that are not functions? Explain.

12.3.2 When conditional assignments are used in designs as comments, the design could be executed if correct actual Pascal statements replaced the comment. Could this be done automatically? That is, could a program be written to take as input a design containing conditional-assignment comments, and create as output correct Pascal?

a) Give a specification for such a program, using the box notation.

b) Explain why such a program *cannot* be written.

12.4 Chapter Summary

A specification is a relation containing those pairs of input and output strings that are acceptable. A program is correct with respect to a specification if for each input represented in the specification relation, the program output agrees with one of the associated specification values.

Different degrees of mathematical formality are appropriate to different situations, but some form of proof of a program's correctness is always appropriate. The proper level of proof depends on the resources available, the audience for the proof, and the importance of the problem and its solution.

Stepwise composition may be alternately viewed as a process of functional refinement, in which top-level functions serve as intentions for lower-level designs, and as specifications against which the implementations can be proved.

PART IV: PROGRAM DESIGN IN D PASCAL

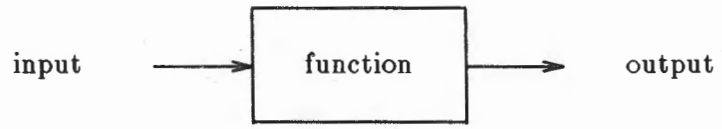
Data Abstractions

Systematic program development in CF Pascal is accomplished through the stepwise refinement of programs. The program calculus and design rules derived from it provide a mathematical basis for writing correct programs in CF Pascal. In Part IV, these design concepts are extended to a more powerful idea, namely that of high-level data and operations in a more powerful language called D (Design) Pascal.

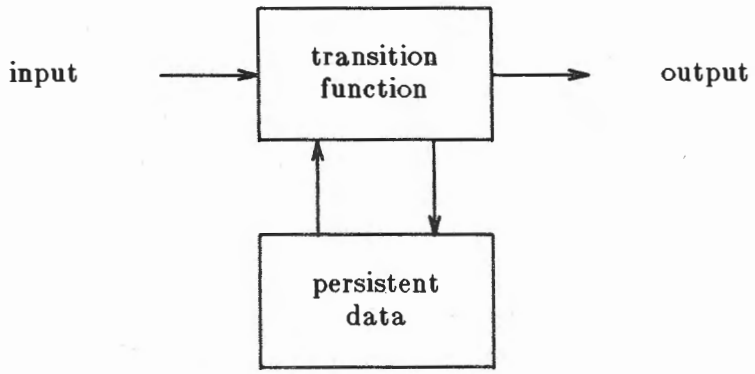
The strategy of stepwise refinement is to solve a problem by defining a small program whose parts are to be designed later. Each part could be later designed as a procedure; the term *procedural abstraction* describes the stepwise-refinement breakdown into subtasks. In any case, whether designed as a Pascal procedure or not, the subprogram implementing the subtask will be a program part with a program part function. Indeed, in stepwise refinement, a program part function is defined to meet the need of the problem being solved, and then a program part, which will have exactly this function, is designed later.

The new design concept in Part IV is *data abstraction.* Whereas a procedure abstraction can be defined by a single Pascal procedure, a data abstraction must be defined by a system of procedures and data declarations, in such a way that these procedures provide the only means of access to this data. Such a system of declarations and procedures is called a *program module.* The advantage of data abstraction over procedure abstraction is that it can be used to retain data for later use, rather than just making calculations as a procedure abstraction does. Files are one data abstraction built into Pascal. Predefined statements (RESET, READ, etc.) manipulate files declared as variables. These statements can be used to store and retrieve data, and the collection of operations and a file declaration defines that file as a data abstraction.

Data abstraction is a more powerful design concept than procedure abstraction because the underlying mathematical idea is a *state machine* rather than a function. A function always produces the same output when presented with the same input:



A state machine can accept an input, produce an output depending on the input and its persistent internal state, and possibly change that internal state:



For example, with a file, consecutive READ statements do not produce the same response. In the file data abstraction, the internal state of the data abstraction will be the file value: a string with cursor and the current file access mode.

D (Design) Pascal

Part IV also extends CF Pascal to a larger subset of Pascal, D Pascal. D Pascal will generally permit smaller and simpler designs for the same part functions than is possible in CF Pascal. The principal extension of D Pascal is in new data types. In addition to the `CHAR` and `TEXT` of CF Pascal, D Pascal contains built-in types `INTEGER`, `BOOLEAN`, `FILE`, `SET`, and `RECORD`. But D Pascal also provides for custom data types of any size and complexity. These custom data types are built up from smaller data types to fit the needs of the problem to be solved. By suitably interpreting the values of a custom data type, it can serve as the basis for a data abstraction. D Pascal also contains additional composite statements and program-structure facilities.

D Pascal is not the full Pascal language, which will appear only in Part V. The subject of this book is the principles of program design, and only incidentally the language Pascal. D Pascal contains the features needed to discuss the important design idea of data abstraction, which is the subject of this part. You will only learn to design programs once, but you will learn several programming languages. Even though Pascal is a good language, it should not create a set of blinders for thinking or programming.

CHAPTER 13

PROGRAM MODULES

Chapter Preview

The most difficult part of programming is maintaining intellectual control over the emerging program, as more and more details are incorporated into it. To maintain control, it is essential that parts of the program be designed and put away, so that they need not be examined in detail again. The program module is a mechanism for encapsulating program ideas in just this way. Several example modules, one to manipulate a counter, others to maintain a queue, will be developed in this chapter. A module differs from a procedure in that it can retain information between successive operations it performs. In this regard it behaves like a formal object called a state machine.

The strategy of stepwise program refinement is to design any program as a hierarchy of design steps, which define tasks, then expand tasks into design parts to meet the task requirements. Each expansion of a task is a program part which will be specified in terms of the part function it must satisfy. This function describes the data transformation performed by the program part.

The main idea in this chapter is to extend the concept of a program part to a new, more powerful design construct, the *program module.* A program module, or more simply a *module,* is defined to be a group of declarations of data and procedures which are specified in a single block of a program with the following restrictions:

> Data declared with the module will be accessed only by the statements (in the procedures) of the module (and by no statements outside the module).
>
> The statements (in the procedures) of the module will access only parameters and data of the module (but no other data outside the module).

A program module provides data storage, processing, and retrieval service for the rest of the program. An illustration of such a service is provided by the declaration of TEXT files and the READ and WRITE statements which manipulate TEXT files. These statements provide the only way to store and retrieve data in the files. Communication between the module and the remainder of the program is provided by the character parameters of the READ and WRITE statements.

The definition of a program module is not explicitly part of the syntax of Pascal, any more than a design part in stepwise refinement is. Rather, it provides another design discipline for organizing programs in an additional, complementary way to stepwise refinement.

The module restrictions permit the module to protect its data from all other parts of the program; that is, once the procedures of the module have been carefully worked out, there will be no danger of the module's data being inadvertently altered or destroyed. This method of defensive programming is called *information hiding,* and the data is said to be *encapsulated* in the module.

This discipline forces all communication with a module to take place through its procedures and their parameters. Therefore, it is not possible for any data in the module to be altered except by a procedure statement that names a procedure of the module. Nor is it possible for such a procedure statement to affect any data outside the module except through the parameters.

A module with only pure storage and retrieval is an important special case. Data enters the module through procedure parameters, and may later be returned unchanged through other parameters. CF Pascal files exemplify this property: the data put into a file always reappears in exactly the same form when it is taken out.

Another important special case of a module is one which has no data declarations, in which pure computation is carried out. A procedure using no global variables satisfies the restrictions all by itself. A group of procedures that form a library, for example, those for natural-number processing suggested in Chapter 9, could constitute a pure-computation module.

Modules provide a new capability for programming design. They allow the invention of data operations at any level needed to solve a problem. For example, if a problem deals with words in a text, modules can be designed to handle the words as single units. Data operations for a crossword puzzle, a chessboard, or a baseball game can all be treated in terms of modules, with data and procedures to deal with the peculiar properties of crossword puzzles, chessboards, or baseball games.

13.1 Numbers

Preview

To illustrate the idea of a module, a three-digit counter will be developed and used.

As an illustration of a simple module that provides both storage/retrieval and computation, consider a programming problem that involves counting up to 999, for example, counting the number of blanks in a string. On certain occasions one needs to increase the counter by one, or to determine the value presently held by the counter, or to reset the counter to zero.

A module can be designed in which the count is maintained in three character variables, as in CountChars in Section 1.6.3:

```
VAR
  Ones, Tens, Hundreds: CHAR
```

and procedures:

```
Start {reset counter to zero}
Bump  {increase counter by one}
Value(VAR V100, V10, V1: CHAR) {return counter value}
```

The counter module is designed so that once Start is executed, the counter holds a three-digit natural number. If the counter holds 999, Bump leaves the counter unchanged. The declaration of the procedures for the counter follow. In an actual Pascal program using this module, the variables within the module would have to be placed in the VAR section.

```
{begin counter module}
{In the VAR section:}
  Ones, Tens, Hundreds: CHAR;
{After the VAR section:}
PROCEDURE Start;
  {reset counter to zero}
  BEGIN {Start}
    Ones     := '0';
    Tens     := '0';
    Hundreds := '0'
  END; {Start}
```

```
PROCEDURE Bump;
  {increase 3-digit counter defined by Hundreds, Tens,
  Ones by one if it is in the range 0 through 999}
  PROCEDURE NextDigit(VAR Digit: CHAR);
    BEGIN {NextDigit}
      IF Digit = '0' THEN Digit := '1' ELSE
      IF Digit = '1' THEN Digit := '2' ELSE
      IF Digit = '2' THEN Digit := '3' ELSE
      IF Digit = '3' THEN Digit := '4' ELSE
      IF Digit = '4' THEN Digit := '5' ELSE
      IF Digit = '5' THEN Digit := '6' ELSE
      IF Digit = '6' THEN Digit := '7' ELSE
      IF Digit = '7' THEN Digit := '8' ELSE
      IF Digit = '8' THEN Digit := '9' ELSE
      IF Digit = '9' THEN Digit := '0'
    END; {NextDigit}
  BEGIN {Bump}
    NextDigit(Ones);
    IF Ones = '0'
    THEN
      BEGIN
        NextDigit(Tens);
        IF Tens = '0'
        THEN
          BEGIN
            NextDigit(Hundreds);
            IF Hundreds = '0'
            THEN
              BEGIN
                Hundreds := '9';
                Ones := '9';
                Tens := '9'
              END
          END
      END
  END; {Bump}
PROCEDURE Value(VAR V100, V10, V1: CHAR);
  {return counter value}
  BEGIN {Value}
    V100 := Hundreds;
    V10 := Tens;
    V1 := Ones
  END {Value}
{end counter module}
```

This declaration and three procedures define a module for a counter which can be inserted into any program in which such a counter is needed. To observe the module restrictions only requires that the rest of the program make no references to the global variables `Ones`, `Tens`, or `Hundreds`. Then the counter is completely hidden behind the three procedures `Start`, `Bump`, and `Value`.

This counter module might be used to solve the problem of counting the number of blanks in an input.

```
PROGRAM CountingBlanksInText(INPUT,OUTPUT);
  VAR
    Ch, X100, X10, X1:  CHAR;
  {include counter module};
  BEGIN {CountingBlanksInText}
    Start; {reset counter to zero}
    WHILE NOT EOF
    DO
      BEGIN
        WHILE NOT EOLN
        DO
          BEGIN
            READ(Ch);
            IF Ch = ' '
            THEN
              BEGIN
                Bump; {increase counter by one}
                Ch := '#'
              END;
            WRITE(Ch)
          END;
        READLN;
        WRITELN
      END;
    WRITELN;
    Value(X100, X10, X1); {get counter value}
    IF (X100='9') AND (X10='9') AND (X1='9')
    THEN
      WRITELN('The number of blanks is at least 999')
    ELSE
      WRITELN('The number of blanks is ', X100, X10, X1)
  END. {CountingBlanksInText}
```

```
INPUT :  Now is
         the  time for
         all good  men.
OUTPUT:##Now#is#
         the##time#for
         all#good##men.

         The number of blanks is 010
```

The procedure statements involving the counter are given extra comments to distinguish them. In the echo, blanks are replaced by # so that they may be seen. The principal cost of using the module in the assembled program is the duplication of variables Hundreds, Tens, and Ones in the module by X100, X10, and X1 in the block outside the module. If the call on Value were removed, X100, X10, and X1 could be removed, and the values of Hundreds, Tens, and Ones printed directly. But this would violate the module restriction that data declared in a module is accessed only by the statements in the module. A few extra variables is a small price to pay for a reusable design module that can be carefully checked and tested once and for all, and used over and over verbatim in other programs. A module library can be even more useful than a procedure library, if the right modules are identified and written.

13.1.1 Exercises

13.1.1 Explain why all three procedures `Start`, `Bump`, and `Value` are needed for a useful counter module by describing the deficiences created by the absence of each.

13.1.2 Consider adding a new procedure called `BumpTen` (to increase the counter by ten) to the counter module.

a) What restrictions must be observed in writing `BumpTen` so that the collection remains a module?

b) Can the text of the new procedure consist of ten procedure statements

```
Bump;
Bump;
 ...
Bump
```

in sequence? Explain why or why not.

c) Write the full text of `BumpTen` without using `Bump`, and show how it is to be incorporated into the module.

13.1.3 Can a module procedure be recursive without violating the access rules?

13.1.4 Suppose a module has just one procedure. Can the data storage for the module be incorporated into the procedure as local variables? Explain.

13.1.5 Create a switch module with variable:

```
VAR
  Switch: CHAR;
```

and procedures:

```
PROCEDURE Set;
  {set switch to down}
PROCEDURE Flip;
  {if the switch is down, set it to up
   if the switch is up, set it to down}
PROCEDURE Value(VAR Sw: CHAR);
  {Sw := 'T' if switch is up, 'F' if down}
```

13.1.6 Use the switch module of the previous exercise to create a program that copies the odd characters of one line of input to the output.

13.1.7 Use the counter module in writing a program to count the number of reversals in a character string. A reversal is three successive characters such that the middle character is either greater or less than the other two characters.

13.1.8 Suppose a certain module contains the following:

```
VAR
  Safe: CHAR;
  {declare other variables of the module here}
FUNCTION Check(VAR OK: CHAR);
  BEGIN {Check}
    IF Safe = 'Y'
    THEN
      OK := 'T'
    ELSE
      OK := 'F'
  END;  {Check}
{other functions and procedures of the module}
```

Users of this module are expected to write code like:

```
Check(Status);
IF Status = 'T'
THEN {call a procedure of the module}
```

so that other procedures of the module can "turn off" its actions by setting `Safe`. A user decides to speed up module usage by writing instead:

```
IF Safe = 'Y'
THEN {call a procedure of the module}
```

a) What rule of module programming is violated by such a user?

b) Explain the purpose of the violated rule, with specific reference to this example.

13.1.9 Design a module for a timer, which keeps time to the nearest minute in days, hours, and minutes, with procedures to reset the timer, add a minute, add an hour, add a day, and return the time. Use the module to calculate elapsed time from an input log of time intervals given in the format *nu* where *n* is a digit and *u* is one of **d**, **h**, or **m** (days, hours, minutes), each pair of intervals separated by a blank. For example, a log entry might be **1h1m□2d2h** with the result 2 days, 3 hours, 1 minute.

13.2 Data Abstractions

Preview

A module may be thought of as introducing a whole new level of objects into a program—the abstract level. In abstract terms, the module contains new, higher-level objects than were available before it was written, and when these objects are used (through the module's operations), none of the lower-level details need be considered.

Because of the data-access discipline in a module, its procedure part functions can be determined entirely from the module itself without reference to the data declared outside. However, two different views need to be considered—those of the module's implementor and those of its users. To work on a module itself, an implementor needs comments that describe the effect of the module's procedures on its hidden data (which is sometimes called its *concrete* data). Since the users of a module cannot reference this hidden data, comments written for them, explaining how to use an operation, must be written in terms of the *abstract* data provided by the module. In the counter module, the implementor's comments would describe the effect of each procedure in terms of `Ones`, `Tens`, and `Hundreds`, while the users' comments would detail the effects of the same operations using an abstract object called `Counter`. `Counter` is not like the other variables in the state in that it was not declared and cannot be directly used in Pascal statements. Instead, the programmer sets the value of `Counter` indirectly via `Start` and `Bump`, and accesses its value using `Value`.

For example, the original `Start` comment:

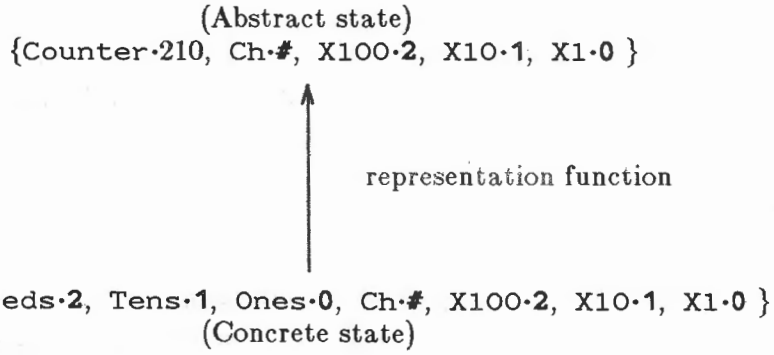
```
{reset counter to zero}
```

can be replaced by the more precise comments:

```
{concrete: Ones,Tens,Hundreds := '0','0','0'}
{abstract: Counter := 0}
```

The first comment is a concurrent assignment that summarizes actual CF Pascal statements. But the second comment uses a variable that is thought of as a natural number, with a natural-number value, not available in CF Pascal. Implementors think in the former terms, users in the latter.

The value of an abstract object can be constructed by mapping the values of the variables used to represent the object to values of the object. This mapping is part of a *representation function* that maps (concrete) states to (abstract) states. A representation function extracts an abstract value from a state and hides the component variables that comprise an abstract object (which is named by a new identifier). For example, in `CountingBlanksInText`, the following states might exist after the call to `Value`:



(INPUT and OUTPUT have been omitted to save space.)

To define the representation function formally, it will be convenient to let *Number* map a list of characters into the natural number it represents in base-ten notation. For example,

Number(<'0', '0', '0'>) = 0 (zero)
Number(<'0', '1', '3'>) = 13 (thirteen).

The representation function R for the counter abstraction above can then be written:

R = {(s, t): t is the same as s except it contains a new identifier Counter, and t(Counter) = *Number*(< Hundreds (s), Tens (s), Ones (s)>) and t does not contain identifiers Hundreds, Tens, Ones}

R maps three character variables to a single integer variable.

The concrete part function for Value is:

```
V100,V10,V1 := Hundreds,Tens,Ones
```

and the abstract part function is:

```
(0<=Counter<=999 → V100,V10,V1 :=
    character representation of hundreds digit of Counter,
    character representation of tens digit of Counter,
    character representation of units digit of Counter)
```

Even though the abstract part function is defined only for digits, there is no requirement in the concrete part function that Hundreds, Tens, and Ones be digits. If Value were executed before Start, the values in Hundreds, Tens, and Ones would be accessed before being set and the function would return unknown values.

The complications in the concrete part function for Bump arise not only from the arithmetic that must be simulated if Ones, Tens, and Hundreds are digits, but from dealing with cases where one or more of these values are not digits.

```
( ('0'<=Ones<'9' → Ones := next(Ones)) |
   (Ones='9' AND '0'<=Tens<'9' → Ones,Tens := '0',next(Tens)) |
   (Ones='9' AND Tens='9' AND '0'<=Hundreds<'9' →
      Ones,Tens,Hundreds := '0','0',next(Hundreds)) |
   (Ones='9' AND Tens='9' AND Hundreds='9' → ) ) |
('0'>Ones OR Ones>'9' → )
```

where *next* is defined as:

next = {(x, y): x and y are digits and y follows x in the sequence **01234567890** }

On the other hand, the abstract comment for Bump is quite compact.

```
(0<=Counter<999 → Counter := Counter + 1 ) |
(Counter < 0 OR Counter >= 999 → )
```

13.2.1 Exercises

13.2.1 Suppose that the counter module is used in a program whose concrete state at some instant is

```
{INPUT·<††,††,R>, OUTPUT·<†So far†,††,W>, NewCh·?, Hundreds·0, Tens·1, Ones·3,
    OldCh·# }
```

a) What will be the abstract state under the representation mapping R?

b) Give the concrete and abstract states following the aliased call:

```
Value(NewCh, NewCh, NewCh)
```

13.2.2 Consider a module and its abstraction called `String5` which contains strings up to length 5, with the operations:

```
PROCEDURE Empty;
  {assign String5 the empty string}
PROCEDURE Append(VAR Ch: CHAR);
  {if String5 not full, add Ch to end of String5}
PROCEDURE Head(VAR Result: CHAR);
  {if String5 empty, Result := ' '
   otherwise String5,Result := tail(String5),head(String5)}
PROCEDURE IsFull(VAR Result: CHAR);
  {if String5 is full, Result := 'Y',
   otherwise Result := 'N'}
```

a) Give the necessary variable declaration(s) and write the text of each procedure.

b) Give concrete and abstract comments for each procedure.

c) What is the representation function for `String5`?

13.3 State Machines

Preview

The action of a program module is like a formal machine with the property that its responses to input depend on its internal state. As each operation is done, the internal state may change, and thus past actions can influence future ones.

A *state machine* is a mechanism (real or imagined) that contains a distinguished state called the *start state*, can accept an input, then produce a new state and an output which depend only on the previous state and input. That is, a state machine is defined by a function that maps a (input, state) pair into an (output, state) pair. For example, a `TEXT` file behaves as a state machine in which the state is the value of the file, and the inputs and outputs are defined by READ and WRITE statements. Taking a file `F1` in execution state s with the value:

$$\boxed{\texttt{F1}}(s) = \texttt{<†AB†, †CD†, R>}$$

and the statement `READ(F1,Ch)` as input to the `TEXT`-file machine, the result is a new state t such that

$$\boxed{\texttt{F1}}(t) = \texttt{<†ABC†, †D†, R>}.$$

The output of the TEXT-file machine could be thought of as the value of `Ch`:

$$\boxed{\texttt{Ch}}(t) = \texttt{C}.$$

The same input again (in state t) will not produce the same result because the internal state is different.

The foregoing suggests the view that execution of a Pascal program can be viewed as the successive actions of a state machine whose state is the execution state of the program. The inputs are the declarations and statements of the program, and there need be no outputs, since full information appears in the program state.

Program modules can be directly modeled as state machines, in the following way:

The names and values of the data objects protected by a module are the states of the machine.

Each procedure statement using a module procedure is an input that causes the machine to change state and (possibly) produce an output.

For example, in the counter module, the state of the counter module state machine is given by the (named) values of `Hundreds`, `Tens`, `Ones`, e.g., as a concrete module state that is a subset of the execution state for the whole program, say:

{`Hundreds`·**3**, `Tens`·**5**, `Ones`·**2**}

(so that the corresponding abstract state contains `Counter`·352). The procedure statement

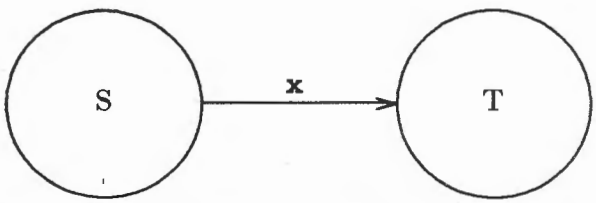
```
Value(V100,V10,V1)
```

is an input that leaves the module state unchanged and returns as output values for `V100`, `V10`, and `V1`, namely **3**, **5**, and **2** respectively. In contrast, the procedure statement `Bump` is an input that creates a new concrete state:

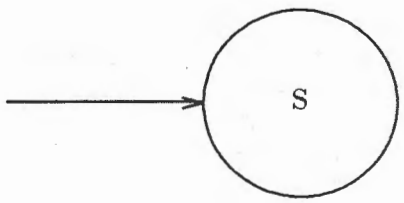
{`Hundreds`·**3**, `Tens`·**5**, `Ones`·**3**}

and produces no output.

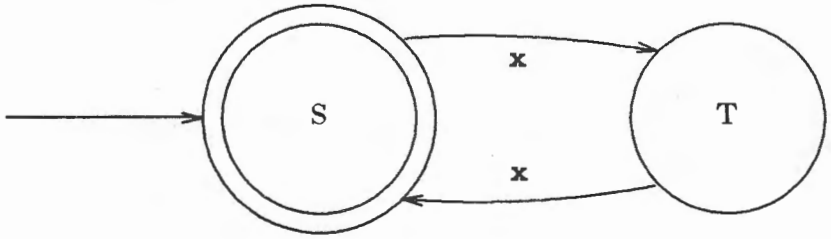
A state machine with a finite number of states is called a *finite state machine.* The machine may be described by a diagram in which each state is represented by a circle containing its name and each transition from one state to another is represented by an arc labeled by the input symbols that cause the machine to change state. In the following diagram, the symbol **x** causes a state change from S to T.



One state of the machine is designated as the *start state*—it will have an unlabeled arc coming into it that does not originate at another state. For example, S is a start state in:

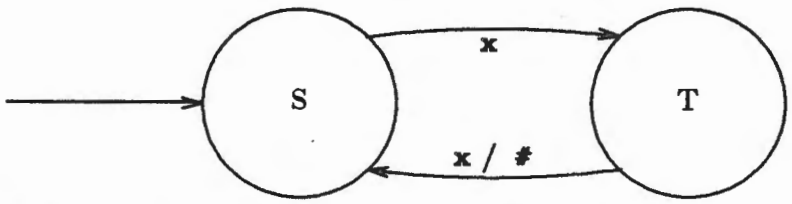


Several states of a finite state machine may be *final states*—they are represented by double circles. The same state can be both the start state and a final state as in the following sample machine.



A machine that begins in its start state, makes a transition for each symbol of the input, and finishes in one of its final states is said to have *recognized* or *accepted* its input. The machine pictured above starts in state S and stops in state S if its input is the empty string or a string consisting of an even number of **x**.

Machines like the one above are called *recognizers* or *acceptors*—they provide a single output: success if they are in a final state at the end of the input, and failure otherwise. Other machines, called *transducers*, may produce an output symbol for each input symbol read. Labeling an arc with symbols separated by / indicates that when the first symbol is read the second symbol is written. The recognizer above can be made into a transducer that outputs half as many **#** as it receives input **x**.



A module with only CHAR internal data, like the counter module, can be modeled by a finite state machine. A module with TEXT internal data cannot be modeled by a finite state machine because there is no bound on the length of a TEXT file, and hence no bound on the number of possible states. Finite state machines are extremely useful for describing situations in which a fixed number of choices arise from input strings of arbitrary length. For example, a finite state machine can be used to determine if an input string of characters contains an ascending run of length at least 10. The state can be the last 9 characters. There are a finite number of such states because the number of possible characters is itself limited. As each input character arrives, the decision can be made that the 9 saved characters plus the arrival do or do not make up the run of 10. If they do the answer is yes; if no run has yet occurred and the string ends, the answer is no. The new state contains the newly arrived character and drops the oldest character. On the other hand, a finite state machine cannot be used to determine if a string is a palindrome by keeping the first half of the string as a state. No matter how large a state is defined, an even larger string may be used as input.

A finite state machine can also help guide program testing. At the minimum, each transition should be explored once by test data. The following BEGIN statement achieves this goal for the counter module.

```
BEGIN {test counter module}
  Start;
  Value(Ch1,Ch2,Ch3);
  WRITELN(Ch1,Ch2,Ch3);
  WHILE (Ch1<>'9' OR Ch2 <>'9' OR Ch3<>'9')
  DO
    BEGIN
      Bump;
      Value(Ch1,Ch2,Ch3);
      WRITELN(Ch1,Ch2,Ch3)
    END;
  Bump;
  Value(Ch1,Ch2,Ch3);
  WRITELN(Ch1,Ch2,Ch3)
END {test counter module}
```

This test tries all the counter values and the transition to the next, but it omits an unlimited number of transitions that the module can make, namely from the **999** state to itself; the test tries only one such transition. Still, if the output has been carefully observed, this seems a good test of the module.

13.3.1 Exercises

13.3.1 Explain how a simple variable of type CHAR can be viewed as a state machine.

13.3.2 Which of the following conditions on a given string can be recognized by a finite state machine?

a) has a unique maximum element;

b) first element greater than last element;

c) has no repeated patterns;

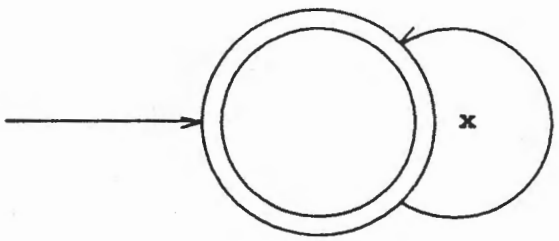
d) has more letters than digits;

e) has more than a million letters.

13.3.3 Create and diagram a recognizer for Pascal identifiers.

13.3.4 Create and diagram a recognizer for Pascal character constants.

13.3.5 What strings does the following finite state machine recognize?



13.4 Queue Modules

Preview

The queue is a useful data structure for holding information in an ordered sequence, much the way people stand in a line. A queue of characters will be made into a module, implemented in two different ways, and used to solve the problem of removing extra blanks from text.

A variable of type TEXT provides a facility for storing and retrieving characters in a sequence using the special syntax of READ/WRITE statements. TEXT variables and operations observe the rules of a program module, because the only way to store or retrieve data in a file is through READ and WRITE.

A *queue* is a similar storage-and-retrieval facility. It operates first-in, first-out (FIFO): elements are added to the rear of a queue and retrieved from the front of a queue. Thus, a queue is like a file except that additions and deletions may be interspersed with one another. For example, consider a queue whose elements are characters. A typical set of operations includes:

Operation	Effect
`EmptyQ`	initialize queue to empty
`AddQ(VAR Elt: CHAR)`	append `Elt` value to queue
`DelQ`	remove first element of queue
`HeadQ(VAR Elt: CHAR)`	set `Elt` to value of queue's first element

If the queue contains one or more elements, `DelQ` removes the first from the queue and `HeadQ` returns the first element in the queue by copying it into `Elt` (but does not alter the queue). If the queue is empty, `DelQ` has no effect and `HeadQ` returns the character #.

In the following sections two module implementations are given for a queue of characters. The first implementation, called `SlowQueue`, is based on a very simple data representation which does not permit much optimization or efficiency in the procedures of the program module. The second implementation (`FastQueue`) uses a more complex data representation and more complex

procedures in order to increase the speed of the operations. Although quite different internally, these two implementations have exactly the same external appearance. The only difference is the time required for execution, not the function or outputs produced. That is, the two modules are interchangeable in any program that uses them. When `FastQueue` replaces `SlowQueue`, no other changes need be made to a program using the module. The program is isolated from the module change because both observe our module rules.

13.4.1 SlowQueue

For a first implementation of a queue consider a strategy of maintaining the queue in a single text file named `Q`:

```
VAR
  Q: TEXT
```

An empty queue is represented as a file containing a single line marker. Since the end of queue is marked in this way, no queue can contain internal line breaks. Each procedure ends with a `RESET` of the file so that when any procedure starts execution it can assume the file is ready for reading. Adding an element to, or deleting an element from the queue, requires WRITE statements. Since the `TEXT` file representing the queue is open for reading, the current contents of the file must be copied to a temporary file and then back to the original file. The outline of the queue module with abstract comments is shown below.

```
{queue module}
VAR
  Q: TEXT;
{include PROCEDURE CopyOpen(VAR Fin, Fout: TEXT)
 Append the future string of Fin to Fout.}
PROCEDURE EmptyQ;
  {Queue := < >}
PROCEDURE AddQ(VAR Elt: CHAR);
  {Queue :=  Queue & <Elt>}
PROCEDURE DelQ;
  {(Queue = < > -->  ) |
    (Queue = <X>&Y --> Queue := Y )}
PROCEDURE HeadQ(VAR Elt: CHAR);
  {(Queue = < > --> Elt := '#' ) |
    (Queue = <X>&Y --> Elt := X )}
{end queue module}
```

The concrete functions corresponding to these abstract functions will now be implemented. `EmptyQ` rewrites `Q` as a single empty line. In the concrete comment, / is used to indicate the end of a line.

```
PROCEDURE EmptyQ;
  {Q := <,/,R>}
  BEGIN {EmptyQ}
    REWRITE(Q);
    WRITELN(Q);
    RESET(Q)
  END  {EmptyQ}
```

`AddQ` copies `Q` to a temporary file and appends the contents of `Elt`. Then the temporary file is copied back to `Q`.

```
PROCEDURE AddQ(VAR Elt: CHAR);
  {Q=<,x/,R> where x is a string AND Elt = a -->
   Q := <,xa/,R> }
  VAR
    Temp: TEXT;
  BEGIN {AddQ}
    REWRITE(Temp);
    CopyOpen(Q,Temp);
    WRITELN(Temp,Elt);
    {copy Temp to Q}
    RESET(Temp);
    REWRITE(Q);
    CopyOpen(Temp,Q);
    WRITELN(Q);
    RESET(Q)
  END  {AddQ}
```

After removing the first character from Q, DelQ copies the remainder of Q to a temporary file, and then copies the temporary back to Q.

```
PROCEDURE DelQ;
  {(Q=<,/,R> --> ) |
   (Q=<,ax/,R> where a is a character and x is a
      (possibly empty) string of characters -->
        Q := <,x/,R>) }
  VAR
    Temp: TEXT;
    Ch: CHAR;
  BEGIN {DelQ}
    {remove first elt from Q};
    READ(Q,Ch);
    IF NOT EOF(Q)
    THEN {not empty}
      BEGIN
        REWRITE(Temp);
        CopyOpen(Q,Temp);
        WRITELN(Temp);
        {copy Temp to Q}
        RESET(Temp);
        REWRITE(Q);
        CopyOpen(Temp,Q);
        WRITELN(Q)
      END;
    RESET(Q)
  END  {DelQ}
```

HeadQ is much like DelQ except that instead of simply removing the first character from Q, HeadQ copies it to Elt.

```
PROCEDURE HeadQ(VAR Elt: CHAR);
  {(Q=<,/,R> --> Elt := '#') |
   (Q=<,ax/,R> where a is a character and x is a
      (possibly empty) string --> Elt := 'a') }
  BEGIN {HeadQ}
    READ(Q,Elt);
    IF EOF(Q)
    THEN
      Elt := '#';
    RESET(Q)
  END  {HeadQ}
```

Another procedure, `WriteQ`, can be added to the queue module for diagnostic purposes. `WriteQ` treats the queue as a list of characters rather than as a queue, writing out one character at a time until end of file is encountered. `WriteQ` has no abstract comment since it is not really an operation on queues, but on raw `TEXT` files. `WriteQ` eases program testing by permitting the contents of `Q` to be observed at any point in the computation. Users need not know how the queue is stored in order to print it, but at the same time `WriteQ` is so simple that it isn't likely to contribute any problems of its own. When program development is finished, `WriteQ` can be commented out of the module.

```
PROCEDURE WriteQ;
  {Q=<,x/,R> and OUTPUT=<y,,W> where x and y are
   (possibly empty) strings --> OUTPUT := <y & x/,,W>}
  BEGIN {WriteQ}
    CopyOpen(Q,OUTPUT);
    WRITELN(OUTPUT);
    RESET(Q)
  END  {WriteQ}
```

13.4.2 FastQueue

In `SlowQueue`, `AddQ` and `DelQ` do too much file copying, degrading the performance of a user's program. The number of copies can be reduced in two ways: (1) using two files and a switch to remember which file holds the current contents of the queue frees us from copying the updated file back into the original file; and (2) using a last-operation indicator permits sequences of just `AddQ`s or just `DelQ`s to be performed without file copies.

The underlying data structure to support this more complex operation must now be more than just the file `Q` of the slow version:

```
VAR
  Q1, Q2: TEXT;
  Switch, LastOp: CHAR;
```

Either `Q1` (if `Switch` has the value **1**) or `Q2` (if `Switch` has the value **2**) holds the contents of the queue. To add an element to the queue whose contents are in the `TEXT` file `Q1`, the contents of `Q1` are copied to `Q2`, a new element is appended to `Q2`, and `Switch` is assigned **2** to indicate that `Q2` rather than `Q1` holds the contents of the queue. A similar sequence of operations can be used to add an element to `Q2`. Thus, the `AddQ` and `DelQ` operations now need only one copy operation instead of the two copy operations that were needed in the slower implementation.

Consecutive applications of either `AddQ` or `DelQ` offer another opportunity to improve the performance of this implementation. `AddQ` copies one file to another and appends the value to the new file. Instead of resetting the new file, `AddQ` can leave it open for writing. If the next operation is also an `AddQ`, the new value can be written on the end of the open file. If an `AddQ` is followed by any other operation, an end marker is written on the file holding the queue's values and the file is reset before the new operation begins. Similarly, successive `DelQ` operations can be implemented by continuing to read from the same file. When an operation other than `DelQ` follows a `DelQ`, the part of the file remaining to be read is copied to the other file before the new operation begins. (The

part of the file already read has been deleted.) LastOp is used to record the last operation so that these rules can be implemented. Its values are:

LastOp	
A	AddQ
D	DelQ
E	EmptyQ
H	HeadQ
W	WriteQ

The operations for this faster version of queue are much more complicated than the corresponding ones for the slow version because the operations use two files instead of one to hold the queue values and these files are not always reset at the end of an operation. EmptyQ, HeadQ, and WriteQ reset the file holding the queue's values so that at the end of one of these operations the file has the form:

$\langle \dagger\dagger, x\ \nabla\ /, \mathbf{R} \rangle$ where x is a string.

Since AddQ leaves the file open for writing, the file has the form:

$\langle x, \dagger\dagger, \mathbf{W} \rangle$ where x is a string.

After DelQ, part of the file may have already been read. This operation leaves the file in the form:

$\langle x, y\ \nabla\ /, \mathbf{R} \rangle$ where x and y are strings.

Each of these possibilities must be considered in writing the operations for the queue.

EmptyQ arbitrarily chooses Q1 to hold the initial value of the queue.

```
PROCEDURE EmptyQ;
  {Q1,Switch,LastOp := <,/,R>,'1','E'}
  BEGIN {EmptyQ}
    REWRITE(Q1);
    WRITELN(Q1);
    RESET(Q1);
    Switch := '1';
    LastOp := 'E'
  END  {EmptyQ}
```

Although WriteQ is merely a testing aid, not part of the final queue module, it will be designed next. The reason is twofold: it will help in testing the other procedures to be developed; and, to write it requires a solid understanding of the representation being used for the queue. In designing WriteQ, the major obstacle to be faced is that the file containing the queue's values can be in any one of three states:

1. $\langle x, y\ \nabla\ /, \mathbf{R} \rangle$ (partially read) if the last operation was DelQ.
2. $\langle \dagger\dagger, y\ \nabla\ /, \mathbf{R} \rangle$ (ready to read) if the last operation was WriteQ, HeadQ, or EmptyQ.
3. $\langle y, \dagger\dagger, \mathbf{W} \rangle$ (partially written) if the last operation was AddQ.

Converting cases 1 and 3 to case 2 greatly simplifies the design task. Since the other queue operations have similar problems, two procedures, CloseDel and CloseAdd, are introduced. CloseDel converts case 1 to case 2, if necessary, and CloseAdd converts case 3 to case 2, if necessary. Therefore, WriteQ begins with CloseDel and CloseAdd. Once the file containing the queue's values is in a state corresponding to case 2, CopyOpen can be used to write the values from Q1 or Q2 to OUTPUT.

```
PROCEDURE WriteQ;
  BEGIN {WriteQ}
    CloseAdd;
    CloseDel;
    LastOp := 'W';
    IF Switch = '1'
    THEN
      BEGIN
        CopyOpen(Q1,OUTPUT);
        WRITELN(OUTPUT);
        RESET(Q1)
      END
    ELSE
      BEGIN
        CopyOpen(Q2,OUTPUT);
        WRITELN(OUTPUT);
        RESET(Q2)
      END
  END  {WriteQ}
```

If the last operation was an AddQ, CloseAdd closes out the appropriate file and resets it. Otherwise CloseAdd does not affect the queue.

```
PROCEDURE CloseAdd;
  BEGIN {CloseAdd}
    IF LastOp = 'A'
    THEN
      IF Switch = '1'
      THEN
        BEGIN
          WRITELN(Q1);
          RESET(Q1)
        END
      ELSE
        BEGIN
          WRITELN(Q2);
          RESET(Q2)
        END
   END  {CloseAdd}
```

WriteQ could be made slightly more efficient by testing to determine if the last operation was an AddQ before executing the procedure statement. However, since this test would have been repeated in many of the other functions, it was placed inside the body of CloseAdd. CloseDel has the same basic structure as CloseAdd—a test to determine if any action is to be taken with the successful outcome resulting in the queue's values being placed in a file that is open for reading.

Design Part 1

```
PROCEDURE CloseDel;
  BEGIN {CloseDel}
    IF LastOp = 'D'
    THEN
      IF Switch = '1'
      THEN
        BEGIN
          Switch := '2';
          {copy remainder of Q1 to Q2 and reset Q2}
        END
      ELSE
        BEGIN
          Switch := '1';
          {copy remainder of Q2 to Q1 and reset Q1}
        END
  END  {CloseDel}
```

The remaining design parts of `CloseDel` are trivial.

Design Part 1.1

```
{copy remainder of Q1 to Q2 and reset Q2}
REWRITE(Q2);
CopyOpen(Q1,Q2);
WRITELN(Q2);
RESET(Q2)
```

Design Part 1.2

```
{copy remainder of Q2 to Q1 and reset Q1}
REWRITE(Q1);
CopyOpen(Q2,Q1);
WRITELN(Q1);
RESET(Q1)
```

`HeadQ` also uses `CloseAdd` and `CloseDel` to reduce the number of possible states of the file containing the queue's values.

Design Part 2

```
PROCEDURE HeadQ(VAR Elt: CHAR);
  {(Switch=1 and x,y are (possibly empty) strings
    of characters and
    ((Q1=<x,y/,R> and LastOp=D) or
     (Q1=<y,,W> and LastOp=A) or
     ((Q1=<,y/,R> and
      (LastOp=W or LastOp=H or LastOp=E))) -->
        Q1, Elt, LastOp := <,y/,R>, head(y & '#'), 'H') |
   (Switch=2 and x,y are (possibly empty) strings
    of characters and
    ((Q2=<x,y/,R> and LastOp=D) or
     (Q2=<y,,W> and LastOp=A) or
     ((Q2=<,y/,R> and
      (LastOp=W or LastOp=H or LastOp=E))) -->
        Q2, Elt, LastOp := <,y/,R>, head(y & '#'), 'H') }
  BEGIN {HeadQ}
    CloseAdd;
    CloseDel;
    LastOp := 'H';
```

```
    IF Switch = '1'
      THEN
        {Elt, Q1 := head(Q1), RESET(Q1)}
      ELSE
        {Elt, Q2 := head(Q2), RESET(Q2)}
  END  {HeadQ}
```

Since the two actions remaining to be developed are so similar, it is natural to think of designing them as procedure statements.

Design Part 2.1

```
{Elt, Q1 := head(Q1), RESET(Q1)}
Head(Q1,Elt)
```

Design Part 2.2

```
{Elt, Q2 := head(Q2), RESET(Q2)}
Head(Q2,Elt)
```

No other program part needs Head, so its declaration can be nested inside that of HeadQ.

Design Part 2.3

```
PROCEDURE Head(VAR Q: TEXT; VAR Ch: CHAR);
  {(Q=<,/,R> --> Elt := '#') |
   (Q=<,ax/,R> where a is a character and
      x is a (possibly empty) string of
      characters --> Elt := 'a') }
  BEGIN {Head}
    IF EOLN(Q)
    THEN
      Ch := '#'
    ELSE
      READ(Q,Ch);
    RESET(Q)
  END  {Head}
```

AddQ either writes its new character value on the end of the current file (if the last operation was an AddQ), or appends it to a copy of the current file made on the other file (if the last operation was not an AddQ).

Design Part 3

```
PROCEDURE AddQ(VAR Elt: CHAR);
  { (LastOp=A -->
       (Switch=1 and Q1=<x,,W> where x is a (possibly empty)
        string of characters and Elt=a -->
           Q1 := <x & 'a',,W> ) |
       (Switch=2 and Q2=<x,,W> where x is a (possibly empty)
        string of characters and Elt=a -->
           Q2 := <x & 'a',,W> ) ) |
    (LastOp<>A -->
       (Switch=1 and either (Q1=<x,y/,R> and LastOp=D)
        or (Q1=<,y/,R> and LastOp<>A and LastOp<>D)
        where x,y are (possibly empty) strings of
        characters and Elt=a -->
           Q1, LastOp := <y & 'a',,W>, 'A') |
       (Switch=2 and either (Q2=<x,y/,R> and LastOp=D)
        or (Q2=<,y/,R> and LastOp<>A and LastOp<>D)
        where x,y are (possibly empty) strings of
        characters and Elt=a -->
           Q2, LastOp := <y & 'a',,W>, 'A') ) ) }
  BEGIN {AddQ}
    IF LastOp = 'A'
    THEN {consecutive AddQs}
      {append Elt to the appropriate file}
    ELSE
      BEGIN
        CloseDel;
        LastOp := 'A';
        {copy one file to the other and append Elt}
      END
  END  {AddQ}
```

Design Part 3.1

```
{append Elt to the appropriate file}
IF Switch = '1'
THEN
  WRITE(Q1,Elt)
ELSE
  WRITE(Q2,Elt)
```

After one file is copied to the other and the contents of `Elt` appended to it, `Switch` must be set to indicate the new file contains the elements in the queue.

Design Part 3.2

```
{copy one file to the other and append Elt}
IF Switch = '1'
THEN
  BEGIN
    Add(Q1,Q2,Elt); {copy Q1 and Elt to Q2}
    Switch := '2'
  END
ELSE
  BEGIN
    Add(Q2,Q1,Elt); {copy Q2 and Elt to Q1}
    Switch := '1'
  END
```

Add is similar to the version of AddQ used in the slow queue except that the file into which values are copied is left open for further writing rather than being reset and copied back into the original file.

Design Part 3.2.1

```
PROCEDURE Add(VAR QFrom, QTo: TEXT; VAR Elt: CHAR);
  BEGIN {Add}
    REWRITE(QTo);
    CopyOpen(QFrom,QTo);
    WRITE(QTo,Elt)
  END  {Add}
```

If the last operation was a DelQ, another DelQ removes another character from either Q1 or Q2, but Switch is not changed since the same file continues to hold the remaining values of the queue.

Design Part 4

```
PROCEDURE DelQ;
 {(LastOp<>D -->
    (Switch=1 and (Q1=<y,,W> or Q1=<,y/,R>) where
     y is a (possibly empty) string of characters -->
        Q1, LastOp := <,tail(y)&'/',R>, 'D') |
    (Switch=2 and (Q2=<y,,W> or Q2=<,y/,R>) where
     y is a (possibly empty) string of characters -->
        Q2, LastOp := <,tail(y)&'/',R>, 'D') ) |
  (LastOp=D -->
    (Switch=1 and Q1=<x,y/,R> -->
       (y is a non-empty string of characters -->
         Q1 := <x comp(head(y)),tail(y)&'/',R> ) |
       (y is the empty string -->  )  ) |
    (Switch=2 and Q2=<x,y/,R> -->
       (y is a non-empty string of characters -->
         Q2 := <x comp(head(y)),tail(y)&'/',R> ) |
       (y is the empty string -->  )  ) ) }
  BEGIN {DelQ}
    IF LastOp = 'D'
    THEN {consecutive DelQs}
      {read first element from appropriate file}
    ELSE
      BEGIN
        CloseAdd;
        LastOp := 'D';
        {copy all but the first element of one file
         to the other}
      END
  END  {DelQ}
```

Design Part 4.1

```
{read first element from appropriate file}
IF Switch = '1'
THEN
  ReadOne(Q1)
ELSE
  ReadOne(Q2)
```

Reading the first element from a file can be packaged as a procedure.

Design Part 4.1.1

```
PROCEDURE ReadOne(VAR Q: TEXT);
  VAR
    Ch: CHAR;
  BEGIN {ReadOne}
    IF NOT EOLN(Q)
    THEN
      READ(Ch)
  END  {ReadOne}
```

When one file is copied to the other, the value of `Switch` must be changed.

Design Part 4.2

```
{copy all but the first element of one file to the other}
IF Switch = '1'
THEN
  BEGIN
    {copy Q1 to Q2 removing first element};
    Switch := '2'
  END
ELSE
  BEGIN
    {copy Q2 to Q1 removing first element};
    Switch := '1'
  END
```

Copying one file to another while removing the first element can be packaged as a procedure.

Design Part 4.2.1

```
{copy Q1 to Q2 removing first element}
Del(Q1,Q2);
```

Design Part 4.2.2

```
{copy Q2 to Q1 removing first element}
Del(Q2,Q1);
```

Design Part 4.2.3

```
PROCEDURE Del(VAR QFrom, QTo: TEXT);
  {copy QFrom to QTo removing first element}
  VAR
    Ch: CHAR;
  BEGIN {Del}
    RESET(QFrom);
    REWRITE(QTo);
    READ(QFrom,Ch);
    CopyOpen(QFrom,QTo);
    WRITELN(QTo);
    RESET(QTo)
  END  {Del}
```

The structure of the fast queue module is shown below. The procedures `Head`, `Add`, `Del`, and `ReadOne` are hidden from the user by nesting their declarations inside the declarations of other procedures. `CloseAdd` and `CloseDel` must remain exposed since `WriteQ`, `HeadQ`, `AddQ`, and `DelQ` must all be able to invoke them.

```
{Include PROCEDURE CopyOpen(VAR FIn, FOut: TEXT)}
PROCEDURE EmptyQ;
PROCEDURE CloseAdd;
PROCEDURE CloseDel;
PROCEDURE WriteQ(VAR FOut: TEXT);
PROCEDURE HeadQ(VAR Elt: CHAR);
   PROCEDURE Head(VAR Q: TEXT; VAR Ch: CHAR);
PROCEDURE AddQ(VAR Elt: CHAR);
   PROCEDURE Add(VAR QFrom, QTo: TEXT;
                 VAR Elt: CHAR);
PROCEDURE DelQ;
   PROCEDURE Del(VAR QFrom, QTo: TEXT);
   PROCEDURE ReadOne(VAR Q: TEXT);
```

13.4.3 RemoveExtraBlanks

If RemoveExtraBlanks as designed in Section 10.1.3 is restricted to processing single lines it can be implemented using a queue. If the text from which blanks are to be removed is already in the queue, then the characters can be deleted from the queue, tested, and either appended to the rear of the queue (if they are not blanks) or discarded (if they are blanks). A special marker must be inserted so the end of the original input text can be detected, and one blank must be inserted between every two words. If the input text is

† **now is the time** †

and the marker is **$**, then the queue initially appears as

.n.o.w. . .i.s. .t.h.e. . .t.i.m.e. .$

(periods are shown to visually separate the 20 queue elements—these are not present in the queue). After 10 elements have been processed from the front, it appears as:

h.e. . .t.i.m.e. .$.n.o.w. .i.s. .t

The design for RemoveExtraBlanks appears below.

Design Part 5

```
PROCEDURE RemoveExtraBlanks;
  {Remove extra blanks between words
   on a single line as held in a queue}
  VAR
    Ch: CHAR;
    Blank, LineEnd: CHAR;
  BEGIN {RemoveExtraBlanks}
    Blank := ' ';
    LineEnd := '$';
    AddQ(LineEnd); {mark end of text in queue}
    HeadQ(Ch);
    {flush blanks}
    WHILE Ch <> LineEnd
    DO
      BEGIN
        {read word};
        {flush blanks};
        {insert blank between words}
      END;
    {remove LineEnd from queue}
  END {RemoveExtraBlanks}
```

The HeadQ(Ch) executed before flushing blanks puts the value of the first queue element in Ch, but does not remove it from the queue. If this value is a blank, it must be removed from the queue and the new value at the head of the queue assigned to Ch.

Design Part 5.1

```
{flush blanks}
WHILE Ch = Blank
DO
  BEGIN
    DelQ;
    HeadQ(Ch)
  END
```

The variable Blank is used instead of the explicit blank character constant; this variable is initialized at the start of the procedure.

Design Part 5.2

```
{read word}
WHILE (Ch <> Blank) AND (Ch <> LineEnd)
DO
  BEGIN
    AddQ(Ch);
    DelQ;
    HeadQ(Ch)
  END
```

Design Part 5.3

```
{insert blank between words}
IF Ch <> LineEnd
THEN
  AddQ(Blank)
```

The fully assembled program is shown below.

```
PROCEDURE RemoveExtraBlanks;
  {Remove extra blanks between words
   on a single line as held in a queue}
  VAR
    Ch: CHAR;
    Blank, LineEnd: CHAR;
  BEGIN {RemoveExtraBlanks}
    Blank := ' ';
    LineEnd := '$';
    AddQ(LineEnd); {mark end of text in queue}
    HeadQ(Ch);
    {flush blanks}
    WHILE Ch = Blank
    DO
      BEGIN
        DelQ;
        HeadQ(Ch)
      END;
    WHILE Ch <> LineEnd
    DO
      BEGIN
        {read word}
        WHILE (Ch <> Blank) AND (Ch <> LineEnd)
        DO
          BEGIN
            AddQ(Ch);
            DelQ;
            HeadQ(Ch)
          END;
        {flush blanks}
        WHILE Ch = Blank
        DO
          BEGIN
            DelQ;
            HeadQ(Ch)
          END;
        {insert blank between words}
        IF Ch <> LineEnd
```

```
        THEN
          AddQ(Blank)
      END;
    DelQ {remove LineEnd from queue}
  END {RemoveExtraBlanks}
```

In order to test RemoveExtraBlanks, it is enclosed in a test driver program which transfers characters from INPUT to a queue, calls RemoveExtraBlanks, and prints the contents of the queue.

```
PROGRAM TestRemove(INPUT,OUTPUT);
  {Reads several lines of input and passes
   them to RemoveExtraBlanks with the line
   markers replaced by blanks}
  VAR
    Ch: CHAR;
  {include either slow queue or fast queue}
  {include PROCEDURE RemoveExtraBlanks}
  BEGIN {TestRemove}
    EmptyQ;
    WRITE('The input is :');
    WHILE NOT EOF
    DO
      BEGIN
        WHILE NOT EOLN
        DO
          BEGIN
            READ(Ch);
            WRITE(Ch);
            AddQ(Ch)
          END;
        READ(Ch);
        IF NOT EOF
        THEN {insert blank between lines}
          BEGIN
           WRITE(Ch);
           AddQ(Ch)
          END
        ELSE {no blank after last line}
          WRITELN
      END;
    RemoveExtraBlanks;
    WRITE('The output is:');
    HeadQ(Ch);
    WHILE Ch <> '#'
    DO
      BEGIN
        WRITE(Ch);
        DelQ;
        HeadQ(Ch)
      END;
    WRITELN
  END.  {TestRemove}
```

Execution

```
INPUT :The input is : Now   is  the time for  all good
OUTPUT:The output is:Now is the time for all good
```

The original input was the three-line string:

```
† Now   is †
†the time for  all†
†good†
```

13.4.4 Exercises

13.4.1 Suppose that the operations `DelQ` and `HeadQ` were combined in `PullQ`:

```
PROCEDURE PullQ(VAR Elt: CHAR)
{Remove first element of queue, and
 return its value in Elt}
```

a) Is this formulation of a queue equivalent to the that of Section 13.4, in the sense that anything one can do, the other can do? Explain.

b) Suppose that `PullQ` is to be *added* to the queue module as another operation. Give the shortest implementation you can for it.

c) Explain why an implementor who decided to *replace* `DelQ` and `HeadQ` with `PullQ`, would not be observing the spirit of the rules for constructing modules.

d) Suppose that an implementation includes `PullQ`, but not `DelQ` or `HeadQ`. Can a module user (not implementor) obtain the exact actions of the latter operations in terms of what exists in the implementation? Explain why not, or else give the statements that will do the job.

13.4.2 Compare the performance of `SlowQueue` and `FastQueue` when included in `RemoveExtraBlanks`, for the input shown at the end of Section 13.4.3. Count the number of read/write operations in each case.

13.4.3 Describe a situation in which `FastQueue` would be *slower* than `SlowQueue`. (Hint: in what sequence of operations would all the tricks `FastQueue` uses be to no avail?)

13.4.4 Using an endline marker in `RemoveExtraBlanks` makes is impossible to handle strings that include this marker as a character.

a) Explain why the endline marker cannot be eliminated by using a test for end of file.

b) Is there a way to avoid wasting a character as an endline marker?

13.4.5 Give an implementation of a queue of characters that holds the queue in five `CHAR` variables, and limits the queue size to five.

13.4.6 Use a queue module to write a procedure that reverses a `TEXT` file. Use diagrams and English description to explain the idea you plan to use, and then do a careful design by stepwise refinement.

13.5 Other Storage and Retrieval Structures

Preview

Queues are not the only useful, simple structures that can profitably be implemented as modules. Sets, stacks, and relations are others.

There are several simple storage and retrieval structures that can be implemented directly with modules: sets, stacks, priority queues, relations, and functions.

A *stack* is a storage and retrieval structure with last-in, first-out (LIFO) access. It can be visualized as a string or list in which elements are inserted and removed from the same end.

A *set* is not ordered, but it can be stored as an ordered list or file in many ways, so long as care is taken in finding, adding, and removing elements no matter where they fall in the order.

The element at the "front" of a *priority queue* is determined first by an arbitrary ordering among elements called priority, and second by time of entry into the queue. Implementing a delete from such a queue involves finding a minimum (or maximum) priority among the members of the queue. One implementation would store the members sorted by priority, but in arrival order within priority.

Relations and *functions* provide storage and retrieval models in which pairs are added to a set-like structure, and elements retrieved by matching the first member of the pair. In the case of a function, a member may not be added if it conflicts with a pair already present. Among the methods of implementing relations and functions is that of maintaining two files, one for the members of the domain and the other for the members of the range, keeping the pairs in the same relative location in the two files. In this way the members of the relation or function can be obtained by reading the files concurrently. However, questions of the range or domain can be handled by dealing with only one file.

13.5.1 Exercises

13.5.1 Discuss the ideas needed to add, remove, and test for elements in a set-of-characters module, implemented as a TEXT file in the order the characters are added, with

a) duplicates eliminated in the file;

b) duplicates entered in the file.

13.5.2 Create and test a module for strings with the following procedures:

```
PROCEDURE MakeString(VAR Raw: TEXT)
{Beginning at the current input position in Raw,
 create the string from the characters up to but
 not including the first  $  character.
 MakeString is to be undefined if no  $  occurs
 in the future string of Raw.  Leave Raw positioned
 following the  $ .}
PROCEDURE LeadingUpto(VAR Marker: CHAR)
{Replace the string with its shortest initial
 substring that ends just before Marker; if no
 Marker value appears, leave the string unchanged.}
PROCEDURE EndingWith(VAR Marker: CHAR)
{Replace the string with the longest substring
 beginning with Marker and extending to the end;
 if Marker does not appear, replace the string with
 the empty string.}
PROCEDURE HeadString(VAR HChar: CHAR)
{If the string has a length of at least 1,
 return its first character in HChar, and
 remove the first character, reducing its length by 1.
 If the string is empty, neither the string nor
 HChar should change and an error message
 should be printed.}
PROCEDURE TailChar(VAR TChar: CHAR)
{Replace the string with another formed by
 adding TChar as an additional, last character.}
```

```
PROCEDURE PrintString
{Write the string to the output (without a newline)}
```

The string should be stored as a TEXT file with appropriate format and discipline. Describe the ideas to be used, particularly the format and discipline of the TEXT file, using English and diagrams, before you begin the design. Then do a careful design, including with each procedure both abstract and concrete comments, being as formal as possible; that is, using string and file mathematical descriptions.

13.5.3 Create a module to manage a priority queue of characters. Use the same operations as those in Section 13.4, but give letters **A-Z** and **a-z** the lowest priority, digits **0-9** the middle priority, and all other characters the highest priority. Follow the suggestions of the previous exercise for documentation.

13.6 Chapter Summary

A program module is a system of procedures and data declarations organized so that the procedures provide the only means of access to the data. Modules provide two views of the data: implementors see concrete data components and users see abstract objects. These views are related by a representation function that combines concrete components into abstract objects, and hides the concrete components. Separating the representation details of abstract objects from algorithms that manipulate the objects permits changes to be made in the representation without affecting the algorithms.

Data abstraction is a more powerful design concept than procedure abstraction because its mathematical abstraction is a state machine rather than a function. A function always produces the same output when presented with the same input. However, a state machine can accept an input then produce an output depending on the input and its internal state, and possibly change its internal state to another state.

CHAPTER 14

ORDINAL DATA TYPES

Chapter Preview

D Pascal extends CF Pascal largely by adding new kinds of objects, data types in addition to characters and files. The ordinal types are those in which the objects are simple rather than compound—each object is an entity, not composed of other objects. An enumerated type contains a fixed, finite collection of named objects. The BOOLEAN type has just the two objects TRUE and FALSE. The INTEGER type is an approximation to the positive and negative whole numbers. Finally, a portion of a type can be made a type in its own right, a subrange type. Of course, the types include operations that define what can be done with the objects. For BOOLEAN, INTEGER, etc., these operations are well known intuitively.

Data types describe sets of values and the operations that can be applied to them. CF Pascal introduced the data types CHAR and TEXT. The values of type CHAR are the set of legal characters of the Pascal machine and the operations on these values are the relational operators:

```
=     <     >     <=     >=     <>
```

whose meanings are the corresponding mathematical operations, with the inequalities signifying alphabetical sequence. The values of the data type TEXT are sequences of lines, each a sequence of characters. The operations on these values are:

```
RESET   REWRITE   READ   WRITE   WRITELN   EOF   EOLN
```

Data types offer the advantages of *abstraction, replication,* and *authentication.* Data types abstract important properties of data. For example, to understand the comparison

```
'A' < 'B'
```

it is not necessary to know how the characters are represented on the Pascal machine—the collating sequence defines the meaning of this operation. Users need not be concerned with the number of bits used to represent a character value, what is done with any extra bits in a word, whether the bit pattern for **B** is exactly one greater than that for **A**, and so on.

Modules achieve a kind of data abstraction, but data types offer something more. Declarations of variables of a data type permit the storage and manipulation of any number of values of the type through these variables. Each use of a variable or constant in a program is authenticated to ensure that only proper operations are applied to it. The context of an operand in a program implies the type of the operand; this implied type is redundant information that can be checked against the declared type of the operand to prevent operations on values of the wrong type. Consider the following program fragment:

```
PROGRAM Typex(INPUT,OUTPUT);
  VAR
    Ch: CHAR;
    F: TEXT;
  BEGIN {Typex}
    ...
    IF Ch = F THEN
    ...
  END.  {Typex}
```

Since = is only defined for character operands in CF Pascal, the appearance of Ch and F as its operands implies that both must have values of type CHAR. While Ch is declared to have the right type, F is declared as TEXT so an inconsistency is detected.

There are two kinds of data types in D Pascal: simple types and aggregate types. Values of simple types do not consist of finer parts that can be separately manipulated. CHAR is a simple type. Values of aggregate types are formed by combining values of other types. TEXT is an aggregate type, because characters are components of its strings.

The values of simple types are ordered, that is, for each pair of values of the type x, y, exactly one of $x < y$, $x = y$, or $x > y$ holds. These are therefore called *ordinal* types. Pascal contains three predefined ordinal types denoted by the identifiers: CHAR, INTEGER, and BOOLEAN. In addition to the predefined ordinal types, Pascal has two methods for programmers to define new ordinal types, namely:

1. Enumerated types whose values are unique Pascal identifiers.
2. Subrange types whose values are some of the consecutive values of another ordinal type.

That is, new ordinal types may be defined by enumerating the constants of the type or by specifying that the values of the type come from an already-defined subrange of values of an existing type. The syntax for denoting these ordinal types is:

<type denoter> ::= <type identifier> | <new type>

<type identifier> ::= <identifier>

<new type> ::= <enumerated type> | <subrange type>

The form for <enumerated type> and <subrange type> will be given in the sections below.

When new types are defined, they can be given names in type definitions. These declarations must precede the variable declarations in a block:

<block> ::= <type definition part> <variable declaration part>
<procedure declaration part> <statement part>

<type definition part> ::= TYPE <type definitions> |

This rule shows that a <type definition part> may be empty (as it has been in all programs up to this point in the text).

<type definitions> ::= <type definitions> <type definition> | <type definition>

<type definition> ::= <identifier> = <type denoter> ;

The context rule that goes with this syntax is that only certain <identifier>s are <type identifier>s:

To be used as a <type identifier>, an <identifier> must have previously appeared in a <type definition>.

Thus in:

```
TYPE
  T1 = CHAR;
  T2 = T1;
```

both T1 and T2 are type identifiers and may be used in the current block to declare variables and formal parameters just as CHAR is used.

14.1 Enumerated Types

Preview

The situation often arises that a small collection of values is needed, each value having a mnemonic name. Instead of using single-character constants, Pascal programmers may employ arbitrary identifiers for these constants.

Any list of Pascal identifiers can be declared as a data type, with the one restriction that no identifier being declared may appear in more than one declaration in a unit of scope. For example, the declarations:

```
TYPE
  DayOfWeek = (Monday, Tuesday, Wednesday, Thursday,
               Friday, Saturday, Sunday);
VAR
  Day: DayOfWeek;
```

defines a new type named DayOfWeek whose seven constant values are written as identifiers naming the days of the week, and a variable named Day that can take any of the constant values. Day may only assume a value from the list given in DayOfWeek. The values of DayOfWeek are ordered as they appear in the type declaration, that is:

```
Monday < Tuesday < ... < Sunday
```

Thus, it is possible to write statements such as:

```
IF Day < Saturday
THEN {not in the weekend}
  ...
```

In the declarations:

```
TYPE
  DayOfWeek = (Sunday, Monday, Tuesday, Wednesday,
               Thursday, Friday, Saturday);
  WeekEndDay = (Saturday, Sunday); {not legal}
VAR
  Friday: CHAR; {not legal}
```

the identifiers of the type WeekEndDay and the variable Friday are illegal duplicates of the identifiers Friday, Saturday, and Sunday that appear first in DayOfWeek. That is, identifiers must be unique over all their uses as names of types, variables, procedures, or as constants in an enumerated type.

Enumerated types are the "supercharacters" of Pascal. They have precisely the same uses as ordinary characters and the same operations. But programmers can invent any number of them, with identifiers that spell out the interpretation of their values. In earlier chapters we used CHAR constants for this purpose, but enumerated types are better. For example, instead of a procedure returning **Y** or **N** to indicate success, and enumerated type with identifiers Yes and No could be used, or even one with identifiers Success and Failure.

Since the letters of the alphabet are identifiers, they can be used as to define a data type:

```
TYPE
  Alphabet = (A,B,C,D,E,F,G,H,I,J,K,L,M,
              N,O,P,Q,R,S,T,U,V,W,X,Y,Z);
VAR
  Letter: Alphabet;
```

This permits statements such as:

```
IF Letter = A
THEN
  Letter := E
```

but not:

```
IF Letter = 'A'
THEN
  Letter := 'E'
```

since A and E are constants of type Alphabet, agreeing with the type of Letter, but 'A' and 'E' are of type CHAR, not the same as the type of Letter.

In many problems the order defined by an enumerated type can be useful. For example, in dealing with dates, the order of the months in the calendar year is significant. While the three-character abbreviations for the months (e.g., Jan, Feb, etc.) are readily recognized and easily used by people, these abbreviations used as strings are ordered lexicographically:

Apr, Aug, Feb, Dec, Jan, Jul, Jun, Mar, May, Nov, Oct, Sep,

which is of no use in calendar questions. However, an enumerated type defined as:

```
TYPE
  Month = (Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);
```

provides an automatic way to compare variables according to the calendar, providing their values are represented as constants of type Month.

14.1.1 Syntax and Meaning of Enumerated Type Expressions

The BNF for enumerated types restricts the constants of the type being defined to identifiers.

<enumerated type> ::= (<identifier list>)

<identifier list> ::= <identifier list> , <identifier> | <identifier>

Context Rule:

The <identifier>s of an <enumerated type> may not duplicate other declared identifiers in a unit of scope.

In particular, a given identifier may be part of just one enumerated type within a unit of scope. This allows a determination of the type of an identifier, and eliminates conflicts of meaning such as that in the DayOfWeek and WeekEndDay example above, where the ordering of Saturday and Sunday would be different in the different types.

Enumerated-type variables and constants may be used in assignment statements and with relational operators, but all the operands used together must be of the same enumerated type. The value of Boolean expressions using relational operators with enumerated-type values is determined by the order in which the values appear in the declaration. For example, in any state s,

$\boxed{A < B}(s)$ = TRUE if and only if $\boxed{A}(s)$ appears before $\boxed{B}(s)$ in the declaration of the type that A and B belong to.

The situation is similar for the other relational operators.

14.1.2 Enumerated Type Input/Output

Unlike CHAR data values which can be read and written by READ and WRITE statements, there is no standard way to read or write enumerated type data. To illustrate how input/output is done, a read/write module will be developed for a type Month representing the months of the year. This module has two procedures, ReadMonth and WriteMonth, but no data declarations. (Thus it is a pure computation module.) ReadMonth converts three characters in a file open for reading into a value of type Month, if possible. For example, if the next three characters are J, A, N, then ReadMonth will deliver the value Jan of type Month. However, if the three characters do not correspond to a value of type Month, an error will be signaled. A simple and useful way to signal an error is to augment Month with one additional value, say NoMonth. Then if the three characters read are not the abbreviation for a month, ReadMonth produces the value NoMonth. The type declaration is:

```
TYPE
  Month = (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec)
```

The module's procedures will most often be used to read from INPUT and write to OUTPUT, but the module discipline requires that these procedures be passed the files they are to read and write;

otherwise they would access data outside the module.

```
PROCEDURE ReadMonth(VAR FIn: TEXT; VAR Mo: Month);
  {FIn.3=R and length(FIn.2)>=3  -->
     read three characters from FIn.2 and assign the
     corresponding Month value to Mo if possible;
     otherwise assign NoMonth to Mo}
  VAR
    Ch1, Ch2, Ch3: CHAR;
  BEGIN {ReadMonth}
    READ(FIn,Ch1,Ch2,Ch3);
    IF (Ch1='J') AND (Ch2='A') AND (Ch3='N')
      THEN Mo := Jan ELSE
    IF (Ch1='F') AND (Ch2='E') AND (Ch3='B')
      THEN Mo := Feb ELSE
    IF (Ch1='M') AND (Ch2='A') AND (Ch3='R')
      THEN Mo := Mar ELSE
    IF (Ch1='A') AND (Ch2='P') AND (Ch3='R')
      THEN Mo := Apr ELSE
    IF (Ch1='M') AND (Ch2='A') AND (Ch3='Y')
      THEN Mo := May ELSE
    IF (Ch1='J') AND (Ch2='U') AND (Ch3='N')
      THEN Mo := Jun ELSE
    IF (Ch1='J') AND (Ch2='U') AND (Ch3='L')
      THEN Mo := Jul ELSE
    IF (Ch1='A') AND (Ch2='U') AND (Ch3='G')
      THEN Mo := Aug ELSE
    IF (Ch1='S') AND (Ch2='E') AND (Ch3='P')
      THEN Mo := Sep ELSE
    IF (Ch1='O') AND (Ch2='C') AND (Ch3='T')
      THEN Mo := Oct ELSE
    IF (Ch1='N') AND (Ch2='O') AND (Ch3='V')
      THEN Mo := Nov ELSE
    IF (Ch1='D') AND (Ch2='E') AND (Ch3='C')
      THEN Mo := Dec
    ELSE Mo := NoMonth
  END;  {ReadMonth}
PROCEDURE WriteMonth(VAR FOut: TEXT; VAR Mo: Month);
  {FOut.3=W and Mo<>NoMonth  -->
     write the three characters corresponding to Mo's
     value to FOut.1}
  BEGIN {WriteMonth}
    IF Mo = Jan THEN WRITE(FOut,'Jan') ELSE
    IF Mo = Feb THEN WRITE(FOut,'Feb') ELSE
    IF Mo = Mar THEN WRITE(FOut,'Mar') ELSE
    IF Mo = Apr THEN WRITE(FOut,'Apr') ELSE
    IF Mo = May THEN WRITE(FOut,'May') ELSE
    IF Mo = Jun THEN WRITE(FOut,'Jun') ELSE
    IF Mo = Jul THEN WRITE(FOut,'Jul') ELSE
    IF Mo = Aug THEN WRITE(FOut,'Aug') ELSE
    IF Mo = Sep THEN WRITE(FOut,'Sep') ELSE
    IF Mo = Oct THEN WRITE(FOut,'Oct') ELSE
    IF Mo = Nov THEN WRITE(FOut,'Nov') ELSE
    IF Mo = Dec THEN WRITE(FOut,'Dec')
  END  {WriteMonth}
```

14.1.3 Comparing Dates

This module can be used to determine the calendar order of two months, given in abbreviation in the input.

Design Part 1

```
PROGRAM CalendarOrder(INPUT,OUTPUT);
  {Recognize (if possible) two three-character abbreviations
   for months and write them out in calendar order}
  TYPE
    Month = (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
             Jul, Aug, Sep, Oct, Nov, Dec);
  VAR
    M1, M2: Month;
  {include module for Month}
  BEGIN {CalendarOrder}
    ReadMonth(INPUT,M1);
    ReadMonth(INPUT,M2);
    {Examine M1 and M2, and write results}
  END.  {CalendarOrder}
```

After both ReadMonth procedure statements have been executed, M1 and M2 each hold a value of type Month (possibly NoMonth). The remainder of the code is straightforward.

Design Part 1.1

```
BEGIN {Examine M1 and M2, and write results}
  IF (M1=NoMonth) OR (M2=NoMonth)
  THEN
    WRITELN('Badly formed input')
  ELSE
    IF M1=M2
    THEN
      BEGIN
        WRITE('Both months are ');
        WriteMonth(OUTPUT,M1);
        WRITELN
      END
    ELSE
      {Compare distinct months M1 and M2,
       and write results}
END
```

Design Part 1.1.1

```
BEGIN {Compare distinct months M1 and M2,
       and write results}
  WriteMonth(OUTPUT,M1);
  IF M1 < M2
  THEN
    WRITE(' precedes ')
  ELSE
    WRITE(' follows ');
  WriteMonth(OUTPUT,M2);
  WRITELN
END
```

Execution

```
INPUT :APRJUL
OUTPUT:Apr precedes Jul
INPUT :AprJul
OUTPUT:Badly formed input
INPUT :JULAPR
OUTPUT:Jul follows Apr
INPUT :DECDEC
OUTPUT:Both months are Dec
```

14.1.4 Exercises

14.1.1 Give a syntax tree for each of the following that is a <type definition>, and for the others explain why each is not.

a) `TYPE Switch = (On, Off);`

b) `TYPE Switch Value = (On, Off);`

c) `TYPE Weekend = (Day1, Day2);`

d) `TYPE Weekend = (1stday, 2ndday);`

14.1.2 Consider a program containing the following type declarations:

```
TYPE
  WeekDay = (Monday, Tuesday, Wednesday, Thursday, Friday);
  WeekEnd = (Saturday, Sunday);
```

Analyze the expression:

a) `Friday < Wednesday`

b) `Friday < Monday`

c) `Monday < Saturday`

14.1.3 Is the following type definition legal? What would be the consequences of permitting enumerated types to be defined in this way?

```
TYPE
  NewAlphabet = ('Z','Y','X','W','V','U','T','S','R',
                 'Q','P','O','N','M','L','K','J','I',
                 'H','G','F','E','D','C','B','A');
```

14.1.4 Create a module for

```
TYPE Switch = (On, Off);
```

with operations `SetOn`, `SetOff`, `Flip`, and `Value` and specifications to match the procedure names. Provide complete instructions for an intended user of the module.

14.1.5 Create an input/output module for enumerated type Color.

```
TYPE
  Color = (Red, Blue, Yellow, Green, Orange);
```

14.1.6 Consider a set of traffic signals at an intersection and a procedure that examines their values and determines if they represent a safe combination. Use the following definitions and declarations.

```
TYPE
  Light = (Red, Yellow, Green);
  Safety = (Safe, Unsafe);
VAR
  North, East, South, West: Light;
  Condition: Safety;
PROCEDURE Traffic(VAR North, East, South, West: Light;
                  VAR Condition: Safety);
  {Determine the safety of the light settings}
```

14.1.7 In the spirit of the previous exercise, create enumerated types, variables, and procedures to analyze a set of traffic signals that have left turn signals as well.

14.2 Type Boolean

Preview

The Boolean operations of NOT, AND, and OR were introduced in Chapter 3. These operations along with variables form the type BOOLEAN of D Pascal. By giving the syntax for Boolean expressions in a somewhat different way, the associativity and precedence of the operators can be specified.

The BOOLEAN type uses the constant values FALSE and TRUE and the operators NOT, AND, OR, and the relationals. The two BOOLEAN constant identifiers act as though they had been declared in the enumerated type declaration:

```
TYPE
  BOOLEAN = (FALSE, TRUE);
```

(But this declaration is not given explicitly.) Thus relational operators are defined on Boolean operands so that the value of

```
FALSE < TRUE
```

is TRUE. NOT is a unary prefix operator and AND and OR are binary infix operators. The functions computed by AND, OR, and NOT are described in Section 3.3.

Pascal guarantees the order in which operators are applied, but not that each operator is applied. Many actual Pascal machines perform "lazy" evaluations of Boolean operators. If the first operand of an OR has value TRUE, the second operand need not be evaluated because

$$\text{TRUE OR } x = \text{TRUE}$$

no matter what x may be. Similarly, if the first operand of an AND has value FALSE, the second operand need not be evaluated because the value of the entire expression will be FALSE. This method of evaluation may cause expressions which would otherwise be undefined to have a value.

Boolean variables can be declared and can be assigned values computed with Boolean expressions. For example, given the declaration:

```
VAR
  EndWord: BOOLEAN;
```

the following Pascal statement assigns either the value TRUE or FALSE to EndWord:

```
EndWord := (Ch = '#') OR (Ch = ' ')
```

14.2.1 Boolean Syntax

The rules of syntax given in Section 3.3 do not cover the case that a BOOLEAN expression may use variables and constants. Those rules also fail to indicate the structure of legal strings as well as they might. New rules for BOOLEAN expressions are shown below.

<expression> ::= <simple expression>
  ¦ <simple expression> <relational operator> <simple expression>

<simple expression> ::= <simple expression> OR <term> ¦ <term>

<term> ::= <term> AND <factor> ¦ <factor>

<factor> ::= <variable access> ¦ <constant> ¦ (<expression>)
  ¦ NOT <factor>

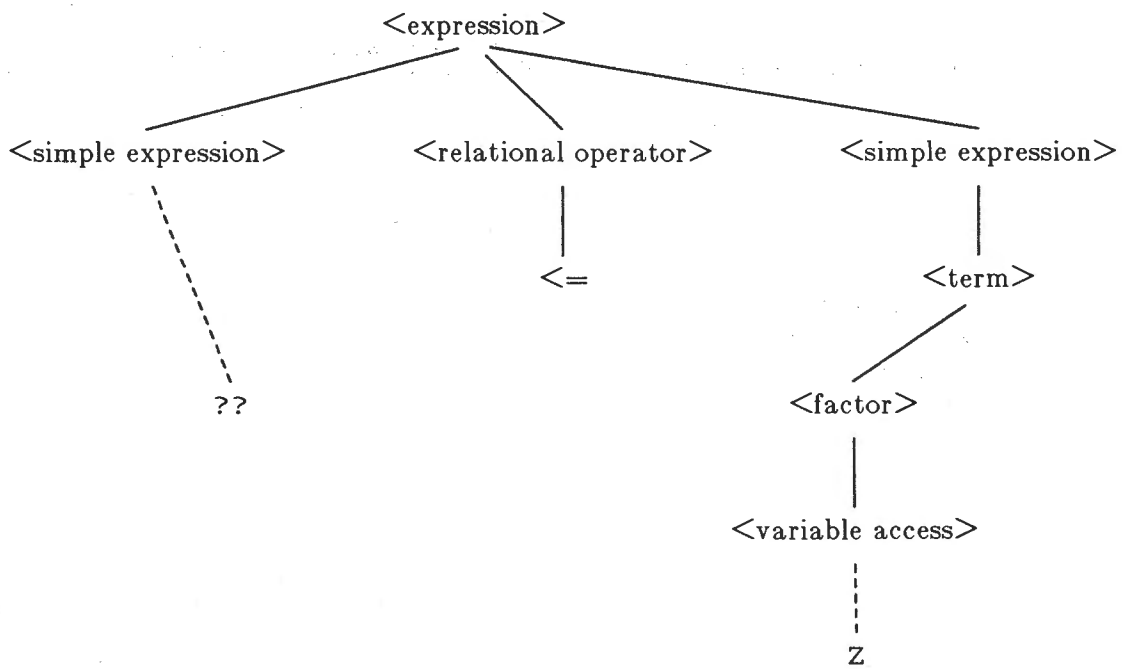
It is not syntactically correct to write

```
X <= Y <= Z
```

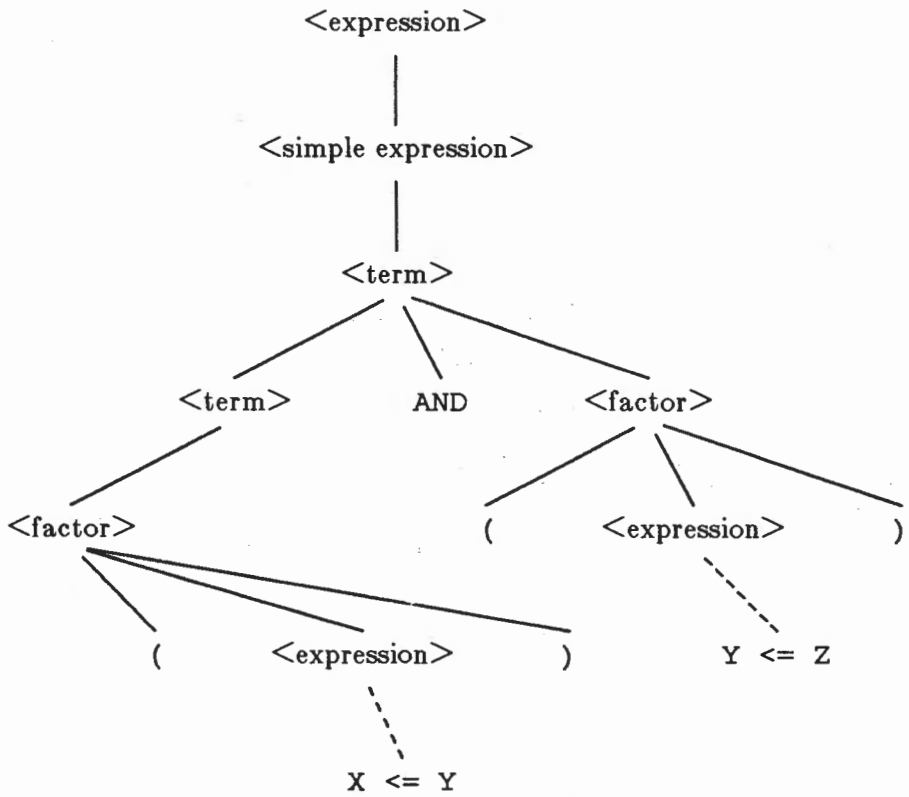
to test that Y has a value between those of X and Z; instead,

```
(X <= Y) AND (Y <= Z)
```

is required to make this test. The parentheses in the expression are part of Pascal's syntax. At attempt to construct a syntax tree for the first expression fails:



It is impossible to derive the string X <= Y from the first <simple expression> without introducing extra characters into the string (e.g., parentheses). However, the other expression can be derived:



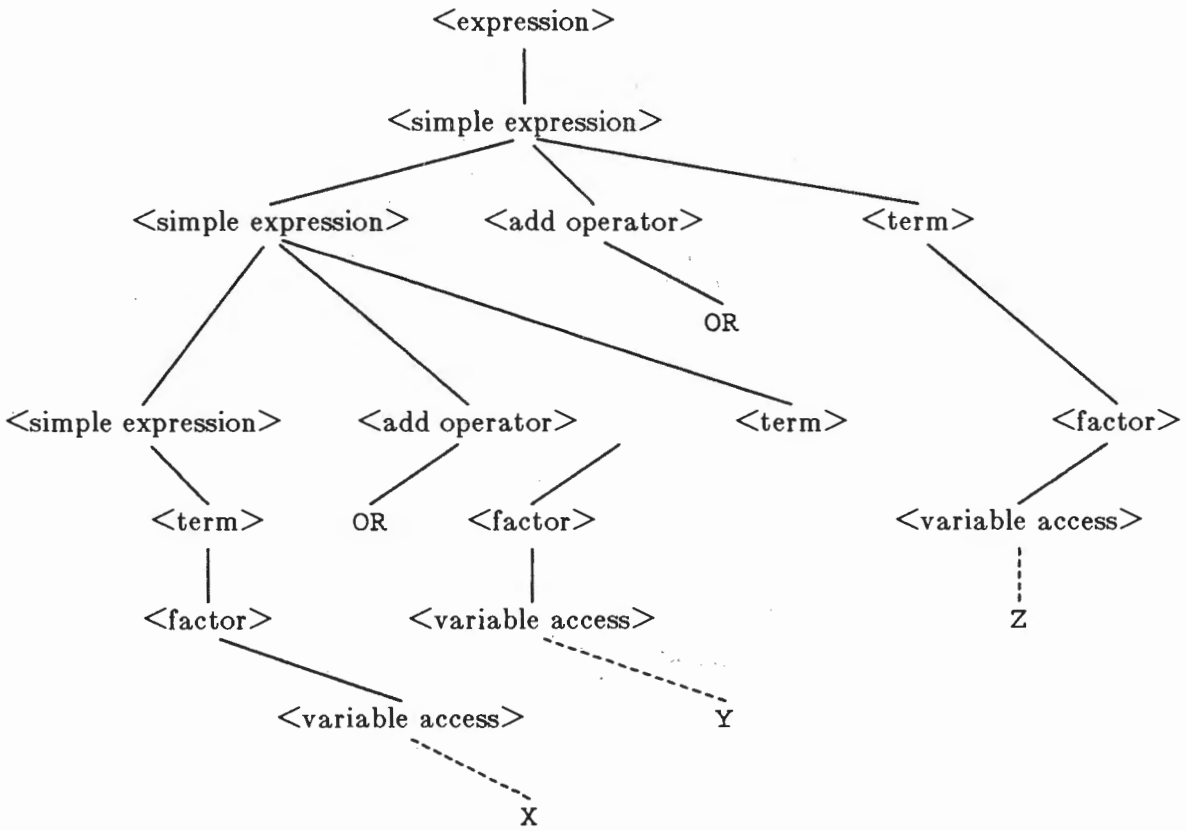
Successive left-associative operators of the same precedence in an expression are applied to their operands in left-to-right order. Thus in the expression:

```
X OR Y OR Z
```

the sub-expression:

```
X OR Y
```

is evaluated first. The syntax tree for this expression is:



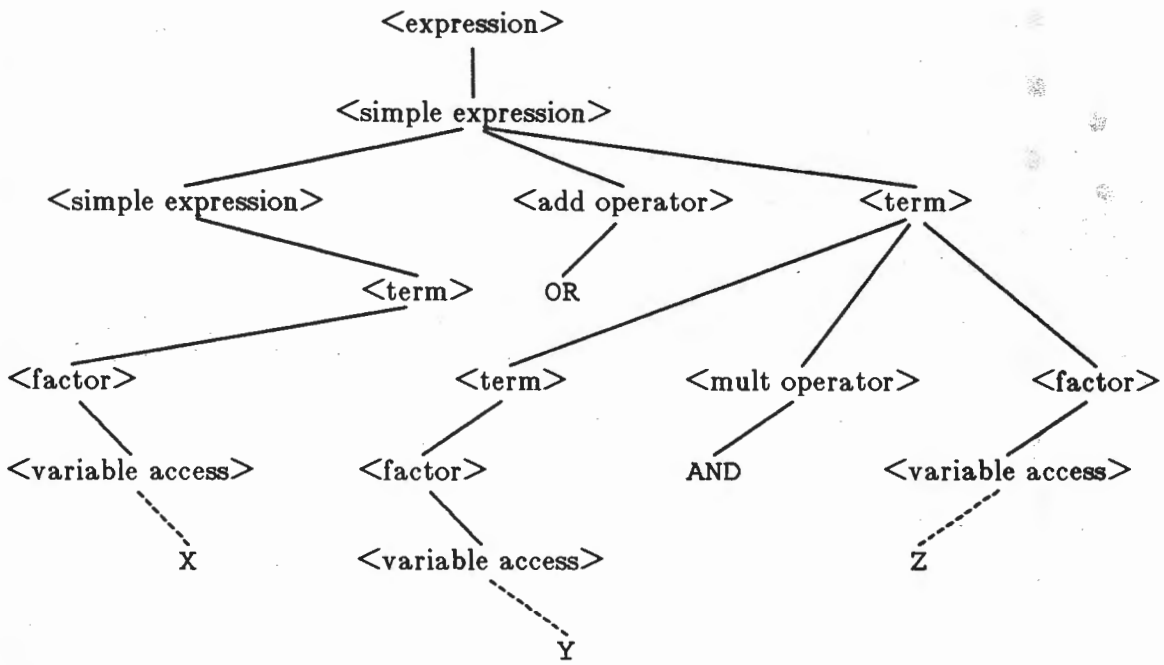
The operands for each OR in the string may be found by locating the strings derived from the corresponding <simple expression> and <term>. The leftmost OR has operands X and Y, while the rightmost OR has operands X OR Y (i.e., the string derived from the first <simple expression> and Z. Thus the leftmost OR is applied to its operands before the rightmost one. (Of course, the precedence of operators in an expression can be altered with parentheses.)

A rule of syntax is *left recursive* if the quantity being defined (i.e., the one to the left of ::=) also appears as the first symbol to the right of ::=. The rule for <term> is left recursive:

<term> ::= <term> AND <factor>

When an operator is defined using left recursion, it will be left associative as shown by the arrangement of its syntax trees.

The relative precedence of operators is also defined by the rules of syntax. The syntax tree for X OR Y AND Z illustrates precedence:



Despite being the rightmost operator in the string, AND can be applied to its operands, Y and Z, before OR can be applied to its operands, X and Y AND Z. Thus AND has a higher precedence than OR in expressions.

14.2.2 The Meaning of Boolean Expressions

The meaning of a Boolean expression is given by the following recursive definition. For any execution state S:

$$\boxed{\texttt{E1 = E2}}(S) = (\boxed{\texttt{E1}}(S) = \boxed{\texttt{E2}}(S))$$

$$\boxed{\texttt{E1 <> E2}}(S) = (\boxed{\texttt{E1}}(S) \neq \boxed{\texttt{E2}}(S))$$

$$\boxed{\texttt{E1 < E2}}(S) = (\boxed{\texttt{E1}}(S) < \boxed{\texttt{E2}}(S))$$

$$\boxed{\texttt{E1 <= E2}}(S) = (\boxed{\texttt{E1}}(S) \leq \boxed{\texttt{E2}}(S))$$

$$\boxed{\texttt{E1 > E2}}(S) = (\boxed{\texttt{E1}}(S) > \boxed{\texttt{E2}}(S))$$

$$\boxed{\texttt{E1 >= E2}}(S) = (\boxed{\texttt{E1}}(S) \geq \boxed{\texttt{E2}}(S))$$

$$\boxed{\texttt{E1 OR E2}}(S) = (\boxed{\texttt{E1}}(S) \text{ OR } \boxed{\texttt{E2}}(S))$$

$$\boxed{\texttt{E1 AND E2}}(S) = (\boxed{\texttt{E1}}(S) \text{ AND } \boxed{\texttt{E2}}(S))$$

$$\boxed{\texttt{NOT E}}(S) = \text{NOT } \boxed{\texttt{E}}(S)$$

$$\boxed{\texttt{(E)}}(S) = \boxed{\texttt{E}}(S)$$

The operators on the left of this definition (e.g., NOT) are those that occur in Pascal programs, while those on the right (e.g., NOT) are mathematical operations. This definition gives the meaning of each operator separately, but does not specify how an expression with multiple operators is to be evaluated. The associativity and precedence of operators in an expression are defined by Pascal's syntax rules. The program calculus requires all operators to be defined; otherwise mathematical properties such as commutativity (which we use more often than we realize) are violated.

14.2.3 Boolean Input/Output

Boolean variables cannot appear in READ statements, but Boolean expressions can appear in WRITE statements. Boolean values are converted to characters and are right-justified in an implementation-defined size (e.g., 10-character) string. The sequence of statements:

```
Condition := TRUE;
WRITE(Condition);
```

appends the string † **TRUE**† to OUTPUT.

14.2.4 Searching a File

The program SarahRevere in Chapter 2 recognizes the words **land** and **sea** in the input. The design parts for this program have been redeveloped using Boolean variables and are reassembled below.

```
PROGRAM SarahRevere(INPUT,OUTPUT);
  VAR
    W1, W2, W3, W4: CHAR;
    Looking, Land, Sea: Boolean;
  BEGIN {SarahRevere}
    BEGIN {Initialize W1, W2, W3, W4,
           Looking, Land, Sea}
      W1 := ' ';
      W2 := ' ';
      W3 := ' ';
      W4 := ' ';
      Looking := TRUE;
      Land := FALSE;
      Sea := FALSE
    END;
    WHILE Looking AND NOT(Land OR Sea)
    DO
      BEGIN
        BEGIN {move window, check end of data}
          W1 := W2;
          W2 := W3;
          W3 := W4;
          READ(W4);
          Looking := W4 <> '#'
        END;
        BEGIN {check window for 'land'}
          Land := (W1='l') AND (W2='a') AND
                  (W3='n') AND (W4='d')
        END;
        BEGIN {check window for 'sea'}
          Sea := (W1='s') AND (W2='e') AND (W3='a')
        END
      END;
```

```
BEGIN {Create Sarah's message}
  IF Land
  THEN
    WRITELN('The British are coming by land.')
  ELSE
    IF Sea
    THEN
      WRITELN('The British are coming by sea.')
    ELSE
      WRITELN('Sarah didn''t say')
END
END.  {SarahRevere}
```

The WHILE statement is controlled by the expression:

```
Looking AND NOT(Land OR Sea)
```

which allows the body of the statement to be executed as long as the value of `Looking` is TRUE and the values of both `Land` and `Sea` are FALSE. Looking is assigned the value FALSE when # is encountered:

```
Looking := W4 <> '#'
```

Land or `Sea` is assigned the value TRUE if `W1`, `W2`, `W3`, `W4` contain the corresponding string, e.g.:

```
Sea := (W1='s') AND (W2='e') AND (W3='a')
```

Boolean variables shorten this program from 57 lines to 43.

14.2.5 Exercises

14.2.1 Draw syntax trees for each of the following, making corrections so that each is syntactically correct. In each case give the order in which the operations are performed, and the declarations needed to make the phrase correct.

a) `NOT NOT A`

b) `A := NOT B > C`

c) `Qu AND (A < B)`

14.2.2 Given declarations

```
TYPE
  Light = (Red, Yellow,Green);
VAR
  North, East, South, West: Light;
```

which describe conditions of traffic lights at an intersection, create a Boolean expression which evaluates to TRUE for safe conditions.

14.2.3 Create a window data type with operations `InitWindow`, `MoveWindow`, and `InspectWindow` so that the `SarahRevere` program in Section 14.2.4 could be written as shown below.

```
PROGRAM SarahRevere(INPUT,OUTPUT);
  ...
  BEGIN {SarahRevere}
    {Initialize Window, Looking, Land, Sea}
    InitWindow;
    Looking := TRUE;
    Land := FALSE;
    Sea := FALSE
    WHILE Looking AND NOT(Land OR Sea)
    DO
      BEGIN
        {move window, check end of data}
        MoveWindow;
        InspectWindow(W1,W2,W3,W4);
        Looking := W4 <> '#';
        {check window for 'land'}
        Land := (W1='l') AND (W2='a') AND
                (W3='n') AND (W4='d');
        {check window for 'sea'}
        Sea := (W1='s') AND (W2='e') AND (W3='a')
      END;
    BEGIN {Create Sarah's message}
      ...
    END
  END.  {SarahRevere}
```

14.3 Type Integer

Preview

The INTEGER data type simplifies programming tasks in which counting is useful. All the usual arithmetic operations are available. Arithmetic using INTEGER operands is fast because most computers have special support for this data type. However, the payment for this speed is a limitation on the magnitude of the integer values that may occur.

The INTEGER data type has values that are positive and negative whole numbers and zero. Each Pascal machine restricts INTEGER values to a subset of the integers:

{-*MAXINT*, ..., -2, -1, 0, 1, 2, ..., *MAXINT*}

MAXINT is a predefined integer constant (typically a power of two) corresponding to the largest integer value that can be represented on a particular Pascal machine. Variables declared to have type INTEGER can be assigned any of these values:

```
VAR
  I, J: INTEGER
```

The INTEGER operators are the relationals and:

```
+       -       *       DIV       MOD
```

for addition, subtraction, multiplication, division with truncation, and remainder on division. Addition, subtraction, and multiplication are subject to overflow (see below), but DIV and MOD are not. The operators + and - can be used both as unary prefix and binary infix operators. Thus the expression J - I could be written -I + J. Division with truncation usually rounds toward zero so that for example:

Operation	Value
5 DIV 3	1
-7 DIV 3	-2
10 DIV -3	-3
-5 DIV -2	2

In the expression I DIV J the result is undefined if the value of J is zero, and zero if the absolute value of I is less than the absolute value of J. Otherwise, the sign of the result is positive if I and J have the same sign, and negative if they have different signs.

The result of the expression I MOD J is undefined if the value of J is less than or equal to zero; otherwise it is the remainder on dividing the value of I by the value of J. For example:

Operation	Value
5 MOD 3	2
-7 MOD 3	-1

Modulus is defined using division with truncation so -7 MOD 3 could be written:

```
-7 - (-7 DIV 3 * 3)
```

The normal rules of arithmetic expression evaluation apply here: operators within parentheses are evaluated first; multiplication and division have precedence over addition and subtraction; and operators of equal precedence are applied left to right. Thus, the value of the previous expression is:

Expression	Evaluation
-7 - (-7 DIV 3 * 3)	start with expression in parentheses
-7 - (-2 * 3)	DIV has the same precedence as *
-7 - (-6)	evaluate expression in parentheses
-1	

The integer operators are summarized below.

Integer Operator Table

Operator	Domain	Range	Associativity	Precedence
unary + -	integer	integer	right	highest
* DIV MOD	integer × integer	integer	left	
+ -	integer × integer	integer	left	
relationals	integer × integer	Boolean	none	lowest

14.3.1 Counting Blanks

In Section 13.1, CountingBlanksInText determined the number of blanks in the input file, using a module to do the counting:

```
PROGRAM CountingBlanksInText (INPUT, OUTPUT);
  VAR
    Ch, X100, X10, X1:  CHAR;
  {include counter module}
  BEGIN {CountingBlanksInText}
    Start; {reset counter to zero}
    WHILE NOT EOF
    DO
      BEGIN
        WHILE NOT EOLN
        DO
          BEGIN
            READ(Ch);
            IF Ch = ' '
            THEN
              Bump; {increase counter by one}
            WRITE(Ch)
          END;
        READLN;
        WRITELN
      END;
    WRITELN;
    VALUE(X100, X10, X1); {get counter value}
    IF (X100='9') AND (X10='9') AND (X1='9')
    THEN
      WRITELN('The number of blanks is at least 999')
    ELSE
      WRITELN('The number of blanks is ',X100,X10,X1)
  END. {CountingBlanksInText}
```

The module defined an abstraction Counter with three component variables. Start assigned the character 0 to each of these variables; Bump simulated the addition of one to the character representation of an integer value; and Value returned the value of the Counter. These procedures added about 50 lines to the program, making it a total of 75 lines long. With the INTEGER data type, Counter can be made an actual program variable:

```
VAR
  Counter: INTEGER
```

and the procedures Start and Bump become assignment statements:

```
Counter := 0 {Start}
Counter := Counter + 1 {Bump}
```

Value is replaced by a reference to Counter, and Counter can be compared to the integer constant 999. The complete program is shown below (now some 25 lines).

```
PROGRAM CountingBlanksInText (INPUT, OUTPUT);
  VAR
    Ch: CHAR;
    Counter: INTEGER;
  BEGIN {CountingBlanksInText}
    Counter := 0; {reset counter to zero}
    WHILE NOT EOF
    DO
      BEGIN
        WHILE NOT EOLN
        DO
          BEGIN
            READ(Ch);
            IF Ch = ' '
            THEN
              Counter := Counter+1; {increase counter by one}
            WRITE(Ch)
          END;
        READLN;
        WRITELN
      END;
    WRITELN;
    IF Counter >= 999
    THEN
      WRITELN('The number of blanks is at least 999')
    ELSE
      WRITELN('The number of blanks is ',Counter)
  END. {CountingBlanksInText}
```

Executions

```
INPUT :  Now is
         the  time for
         all good  men.
OUTPUT:  Now is
         the  time for
         all good  men.

         The number of blanks is          9
INPUT :
OUTPUT:
         The number of blanks is          0
```

The test for more than three-digit values was retained to make this program look like the one with the Counter module; with an INTEGER variable the proper limitation is *MAXINT* instead. It is a common (unsafe) practice to ignore *MAXINT* and hope that the Pascal machine will deal with any problems that arise.

14.3.2 Integer Input/Output

The output from CountingBlanksInText looks somewhat strange—many blanks separate the end of the string from the start of the integer value. When integer values are written into TEXT files, they are converted to characters and placed right-adjusted in a fixed-length string (frequently 10 characters long). Leading zeros are removed. Notice the difference between the statements WRITE('5') and WRITE(5). The first statement appends the string †5† to OUTPUT, while the second adds the string

```
†    5†
```

because the constants are of type CHAR and INTEGER, respectively.

Pascal has formatting capabilities to control the size of the field in which an INTEGER value appears right justified. Any of the integer expressions of a WRITE statement can be suffixed with a colon and an integer expression that indicates the size of field for the expression. For example, if X and Y are integer variables with the values 123 and 5, respectively, then WRITE(X:Y) appends the string † **123**† to OUTPUT. The new syntax rules for the WRITE statement are:

<writeln parameter list> ::= (<file variable>) ¦ <write parameter list> ¦

<write parameter list> ::= (<write item list>)

<write item list> ::= <file variable> <write parameters>
¦ <write parameters>

<write parameters> ::= <write parameters> <write parameter>
¦ <write parameter>

<write parameter> ::= <expression> : <expression> ¦ <expression>

Successive integer values can be read from a TEXT file if they are separated by blanks or line markers. If Int1 is an INTEGER variable, the statement READ(Int1) will expect to find and read a sequence of characters from INPUT that form a signed integer. The string of characters is converted to an integer value before being stored in Int1. The syntax for these character sequences in INPUT is the same as for INTEGER constants in Pascal programs:

<signed integer> ::= <sign> <unsigned integer> ¦ <unsigned integer>

<sign> ::= + ¦ -

<unsigned integer> ::= <digit sequence>

<digit sequence> ::= <digit sequence> <digit> ¦ <digit>

<digit> ::= 0 ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 ¦ 8 ¦ 9

When an INTEGER variable appears in a READ statement, blanks and line markers (if necessary) are skipped in the file until a nonblank character is reached. The READ operation consumes those characters that can be part of an integer constant including leading signs, and stops just before the first character that cannot be part of the constant. The integer value corresponding to the characters consumed is stored in the variable. For example, given the following string in INPUT:

```
†  -12 3/32/†
```

where / represents the line marker, and the program fragment:

```
VAR
  I1, I2, I3: INTEGER;
BEGIN
  READ(I1, I2, I3);
  ...
```

the execution of the READ statement will produce a state with partial contents:

{I1-12, I2-3, I3-32}.

When reading a TEXT file containing an unknown number of integers, the final line marker remains after the last integer is read. Thus EOF cannot be used as a test for termination, as the program ReadInts mistakenly attempts to do:

```
PROGRAM ReadInts(INPUT,OUTPUT);
  {A mistaken attempt to read all the
   integers from the input}
  VAR
    Val: INTEGER;
  BEGIN {ReadInts}
    WHILE NOT EOF DO
      BEGIN
        READ(Val);
        WRITELN(Val)
      END
  END.  {ReadInts}
```

After the last integer is read, the line marker remains, and `EOF` is still FALSE; hence the program reads again, and there being no integer value, the READ statement is undefined. The following is better:

```
PROGRAM ReadInts2(INPUT,OUTPUT);
  {Read a file of integers with
   attention to the final line marker}
  VAR
    Val: INTEGER;
  BEGIN {ReadInts}
    WHILE NOT EOF DO
      BEGIN
        IF EOLN
        THEN
          READLN;
        IF NOT EOF
        THEN
          BEGIN
            READ(Val);
            WRITELN(Val)
          END
      END
  END.  {ReadInts}
```

However, even this program will fail if the final line mark does not immediately follow the last integer, for example, on the input string †**123x**† or †**123** †. It is easier to read integers from a `TEXT` file using a sentinel value to indicate when the input data is exhausted (e.g., **-99999** or **0**, some value that cannot legitimately appear in the file).

To illustrate computation with integers, the following program obtains an integer N from the input, then computes the sum of the first N integers:

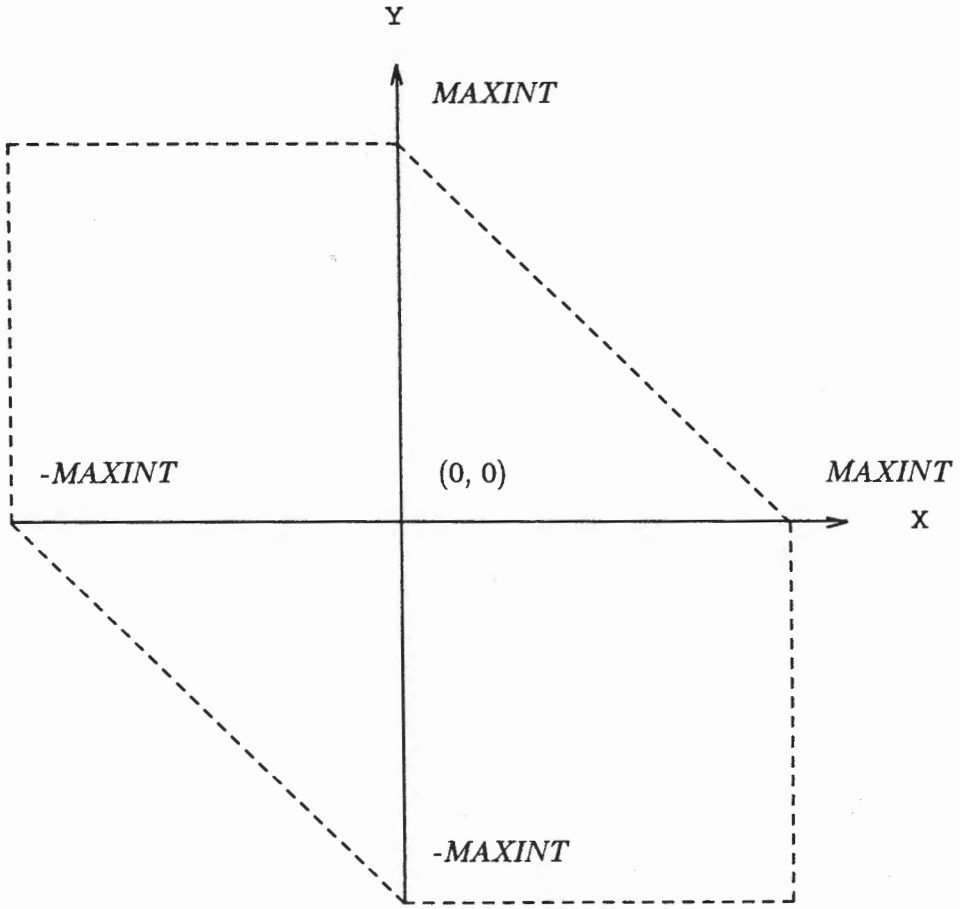
```
PROGRAM Sum(INPUT,OUTPUT);
  { (N>=0 --> I,S := N+1,1+2+...+N) }
  VAR
    I, N, S: INTEGER;
  BEGIN {Sum}
    READ(N);
    S := 0;
    I := 1;
    WHILE I <= N
    DO
      BEGIN
        S := S + I;
        I := I + 1
      END;
    WRITELN('The sum of the first',N:2,
            ' positive integers is',S:3)
  END.  {Sum}
```

Execution

```
INPUT :6
OUTPUT:The sum of the first 6 positive integers is 21
```

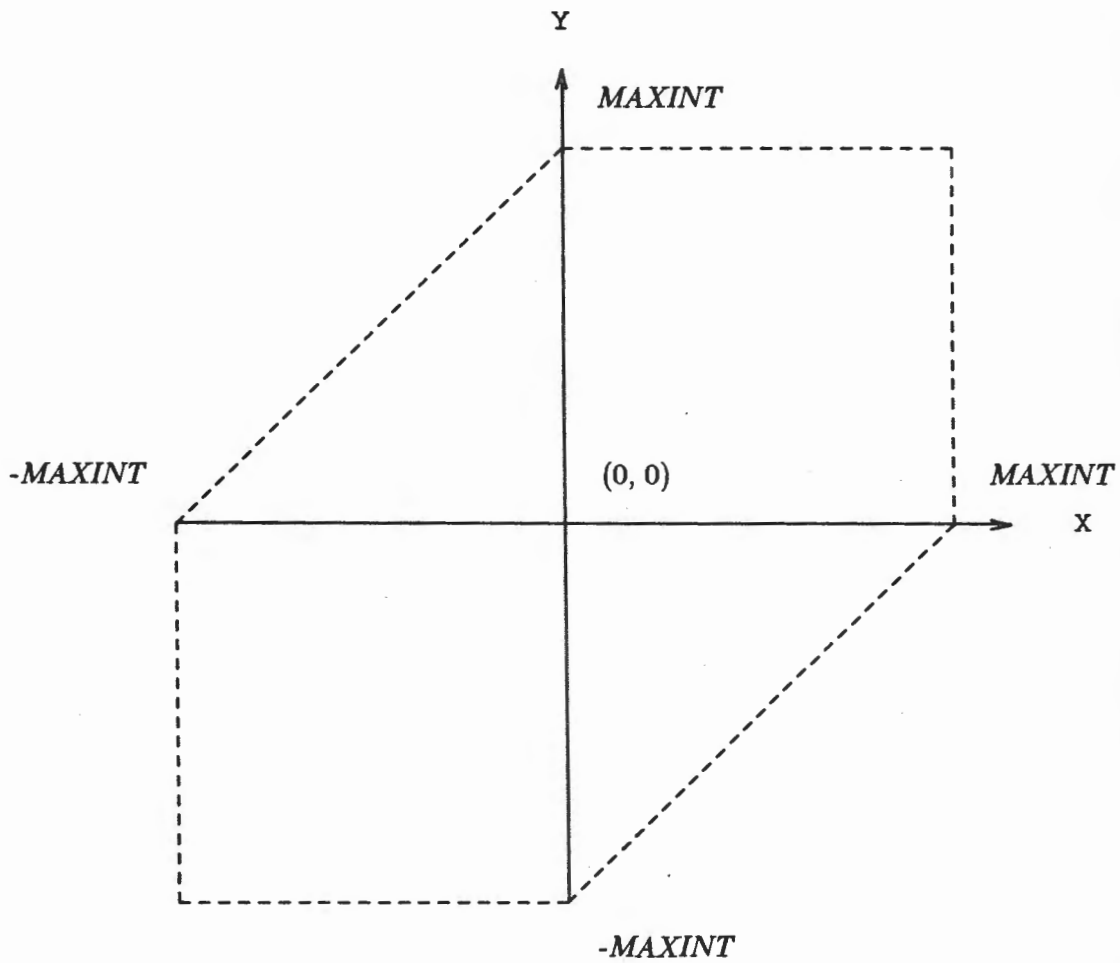
14.3.3 Overflow

Pascal `INTEGER` operations are only guaranteed to produce a correct result if all the values involved lie within the range [*MAXINT*, *MAXINT*]. That is, the operands must be in range, and the result of the operation as well. For the addition operation, the figure shows the region of correct results for `X + Y` within the dashed lines.



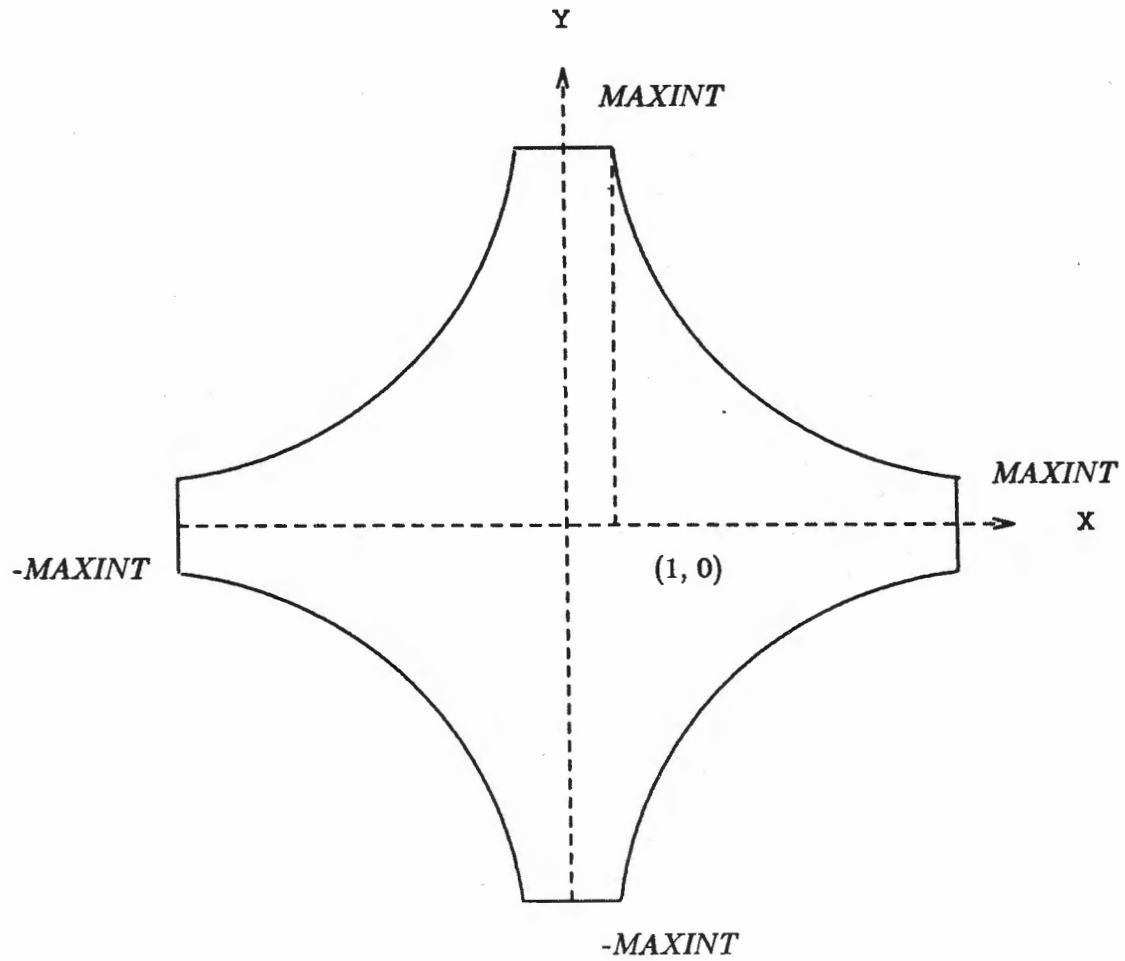
Outside the boundaries defined by *MAXINT*, addition does not produce correct results. Starting with operands chosen at random within range, there is 1 chance in 4 that the result will overflow (1/4 of the area bounded by $\pm$*MAXINT* in the figure represents error results).

Similarly, the region for correct results for subtraction `X - Y` is shown in the figure:



Again there is probability .25 that overflow will occur in subtracting two random INTEGER values.

Since multiplication can generate large values more easily than addition or subtraction, the overflow problem is even more dramatic for this operation. The region of correct results for multiplication X*Y is shown below by the solid lines:



The curved boundaries are hyperbolas, and the figure distorts the scale to show unit distance from each axis. In fact the area of correct results is surprisingly small. In one quadrant, the area under the curve $Y = MAXINT/X$ is given by:

$$MAXINT + \int_{1}^{MAXINT} \frac{MAXINT}{X}\, dX$$
$$= MAXINT + MAXINT \ln MAXINT - MAXINT \ln 1.$$

Thus the area under the curve is $MAXINT + MAXINT \ln MAXINT$, while the entire area is $MAXINT^2$, so the chance of randomly selecting two operands for which overflow will not occur is:

$$\frac{MAXINT + MAXINT \ln MAXINT}{MAXINT^2} = \frac{1 + \ln MAXINT}{MAXINT}.$$

For $MAXINT=10^9$, the chance of correct results from the multiplication of values chosen at random from type INTEGER is

$$\frac{1 + \ln 10^9}{10^9} \approx \frac{50}{10^9},$$

or about 1 in 20 million.

In contrast to addition, subtraction, and multiplication, the division operators DIV and MOD cannot cause overflow.

It is clear that type INTEGER represents only a small-scale, partial model of mathematical integers. As long as the limitations are observed, the operations are perfect, but it is the responsibility of the programmer, not the Pascal machine, to honor these limitations. It is a defect of Pascal that the value of *MAXINT* must be determined by trial for each different machine, so the programmer who wants to be careful cannot do so easily. On some machines, computation is aborted when overflow occurs, so it is safe to ignore its possibility as long as a surprise program termination is not important. On many machines, however, the results simply stop conforming to the expected arithmetical rules when overflow occurs.

14.3.4 Integer Syntax

The updated syntax rules for Pascal <expression> are shown below.

<expression> ::= <simple expression>
  ¦ <simple expression> <relational operator> <simple expression>

<simple expression> ::= <simple expression> <add operator> <term>
  ¦ <term> ¦ <sign> <term>

<term> ::= <term> <mult operator> <factor> ¦ <factor>

<factor> ::= <variable access> ¦ <unsigned constant>
  ¦ (<expression>) ¦ NOT <factor>

<relational operator> ::= = ¦ <> ¦ < ¦ <= ¦ > ¦ >=

<sign> ::= + ¦ -

<add operator> ::= + ¦ - ¦ OR

<mult operator> ::= * ¦ DIV ¦ MOD ¦ AND

<variable access> ::= <variable identifier>

<unsigned constant> ::= <unsigned number> ¦ <constant identifier>
  ¦ <character string>

<unsigned number> ::= <unsigned integer>

14.3.5 Meaning of Integer Expressions

The meaning of an integer expression is given by the following recursive definition for state S:

$[\![\texttt{E1 = E2}]\!](S) = ([\![\texttt{E1}]\!](S) = [\![\texttt{E2}]\!](S))$

$[\![\texttt{E1 <> E2}]\!](S) = ([\![\texttt{E1}]\!](S) \neq [\![\texttt{E2}]\!](S))$

$[\![\texttt{E1 < E2}]\!](S) = ([\![\texttt{E1}]\!](S) < [\![\texttt{E2}]\!](S))$

$[\![\texttt{E1 <= E2}]\!](S) = ([\![\texttt{E1}]\!](S) \leq [\![\texttt{E2}]\!](S))$

$[\![\texttt{E1 > E2}]\!](S) = ([\![\texttt{E1}]\!](S) > [\![\texttt{E2}]\!](S))$

$[\![\texttt{E1 >= E2}]\!](S) = ([\![\texttt{E1}]\!](S) \geq [\![\texttt{E2}]\!](S))$

$[\![\texttt{E1 + E2}]\!](S) = ([\![\texttt{E1}]\!](S) + [\![\texttt{E2}]\!](S))$ (subject to overflow)

$[\![\texttt{E1 - E2}]\!](S) = ([\![\texttt{E1}]\!](S) - [\![\texttt{E2}]\!](S))$ (subject to overflow)

$[\![\texttt{E1 * E2}]\!](S) = ([\![\texttt{E1}]\!](S) \times [\![\texttt{E2}]\!](S))$ (subject to overflow)

$$\boxed{\texttt{E1 DIV E2}}(S) = (\boxed{\texttt{E1}}(S) \div \boxed{\texttt{E2}}(S))$$

$$\boxed{\texttt{E1 MOD E2}}(S) = (\boxed{\texttt{E1}}(S) - (\boxed{\texttt{E1}}(S) \div \boxed{\texttt{E2}}(S) \times \boxed{\texttt{E2}}(S)))$$

$$\boxed{\texttt{+E}}(S) = \boxed{\texttt{E}}(S)$$

$$\boxed{\texttt{-E}}(S) = -\boxed{\texttt{E}}(S)$$

$$\boxed{\texttt{(E)}}(S) = \boxed{\texttt{E}}(S)$$

The definition gives the meaning of each operator. For an expression with more than one operator, the meaning is built up term by term, according to Pascal's syntax rules for precedence and associativity.

14.3.6 Converting Characters to Numbers

Consider a string of digits of length N:

$$c_1 c_2 \ldots c_{N-1} c_N$$

The correspondence between the string and the number it represents is given by the formula: $\sum_{i=1}^{N} (c_i \times B^{N-i})$ where B stands for the base and c is the numeric value of the i-th character in the string. Thus the number 0101 stands for the following base-ten values in the indicated bases:

$$0101_2 = 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 = 0 + 4 + 0 + 1 = 5_{10}$$
$$0101_8 = 0\times8^3 + 1\times8^2 + 0\times8^1 + 1\times8^0 = 0 + 64 + 0 + 1 = 65_{10}$$
$$0101_{10} = 0\times10^3 + 1\times10^2 + 0\times10^1 + 1\times10^0 = 0 + 100 + 0 + 1 = 101_{10}$$
$$0101_{16} = 0\times16^3 + 1\times16^2 + 0\times16^1 + 1\times16^0 = 0 + 256 + 0 + 1 = 257_{10}$$

The "digits" of bases greater than ten are often represented by letters, e.g., the digits of base sixteen are:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

The best strategy to find the base-ten value of the base-B expression:

$$c_1 \times B^{N-1} + c_2 \times B^{N-2} + \ldots + c_N \times B^0$$

is to factor it using Horner's rule:

$$(\ldots(c_1 \times B + c_2) \times B + c_3) \ldots \times B + c_N).$$

The program `StringToInt` converts strings of characters in the base specified in Base to numeric values in Result using Horner's rule.

Design Part 1

```
PROCEDURE StringToInt(VAR F: TEXT; VAR Base,Result: INTEGER);
  {F=<,x#y,R> where x is a string of digits and
   Base is an integer, 2<=Base<=16, indicating
   their base --> F,Result := <x#,y,R>,N(x)
   where N(x) is the integer value of the character
   representation x}
  VAR
    Ch: CHAR;
    Digit: INTEGER;
  BEGIN {StringToInt}
    Result := 0;
    READ(F,Ch);
    WRITE(Ch);
    WHILE (Ch <> '#') AND (Ch <> ' ')
    DO
```

```
    BEGIN
      {Digit := number corresponding to Ch in Base};
      Result := Result*Base + Digit;
      READ(F,Ch);
      WRITE(Ch)
    END
  END;  {StringToInt}
```

The code to find the integer corresponding to Ch is straightforward but long, so it is packaged in a procedure.

Design Part 1.1

```
{Digit := number corresponding to Ch in Base}
CharToDigit(Ch,Digit);
```

In this problem, only StringToInt references CharToDigit so the latter's declaration can be placed inside the former's declaration.

```
PROCEDURE CharToDigit(VAR Ch: CHAR;
                      VAR Result: INTEGER);
  {('0'<=Ch<='9' OR 'A'<=Ch<='F' -->
      Result := number corresponding to Ch) |
   (NOT('0'<=Ch<='9' OR 'A'<=Ch<='F') -->
      Result := 0)}
  BEGIN {CharToDigit}
    IF Ch = '0' THEN Result := 0 ELSE
    IF Ch = '1' THEN Result := 1 ELSE
    IF Ch = '2' THEN Result := 2 ELSE
    IF Ch = '3' THEN Result := 3 ELSE
    IF Ch = '4' THEN Result := 4 ELSE
    IF Ch = '5' THEN Result := 5 ELSE
    IF Ch = '6' THEN Result := 6 ELSE
    IF Ch = '7' THEN Result := 7 ELSE
    IF Ch = '8' THEN Result := 8 ELSE
    IF Ch = '9' THEN Result := 9 ELSE
    IF Ch = 'A' THEN Result := 10 ELSE
    IF Ch = 'B' THEN Result := 11 ELSE
    IF Ch = 'C' THEN Result := 12 ELSE
    IF Ch = 'D' THEN Result := 13 ELSE
    IF Ch = 'E' THEN Result := 14 ELSE
    IF Ch = 'F' THEN Result := 15
    ELSE
      BEGIN
        Result := 0
      END
  END;  {CharToDigit}
```

To test this design, it is placed in a program that first reads a base-ten representation of the base in which the following string is to be interpreted.

```
PROGRAM Convert(INPUT,OUTPUT);
  VAR
    Base, Number: INTEGER;
  {Include PROCEDURE StringToInt
            (VAR F: TEXT; VAR Base, Result: INTEGER);
   ( F=<,i□x#y,R> where i is a base-ten integer
   between 2 and 16 and x is a string of digits
      --> F,Number := <i□x#,y,R>,N(x) )
   where N(x) is the base-i value of x }
  BEGIN {Convert}
    READ(Base);
    StringToInt(INPUT, Base, Number);
    WRITELN;
    WRITELN('The base-ten value is:', Number)
  END.  {Convert}
```

Executions

```
INPUT :8 0101#
OUTPUT:8 0101#
       The base-ten value is:        65
INPUT :16 1F#
OUTPUT:16 1F#
       The base-ten value is:        31
```

14.3.7 Exercises

14.3.1 What is the value of *MAXINT* in your Pascal machine? What happens when you try to read a value larger than *MAXINT* from the input? What happens when you try to write a constant larger than *MAXINT* as a part of a Pascal program? What happens when you try to add two integers whose sum exceeds *MAXINT*?

14.3.2 Suppose that the expression

```
(X + Y*Z) DIV W
```

using INTEGER variables X, Y, Z, W occurs in a program, and the programmer wishes to be certain that no overflow ever occurs when it is calculated. Given that the machine being used has a value of *MAXINT* held in the INTEGER variable MI, write appropriate IF statements to protect this expression.

14.3.3 In Section 14.3.2 a program ReadInts2 was written to read a series of integers from the input, being careful about line markers. That program cannot handle the situation in which the final integer is followed by extraneous characters.

a) Explain exactly what ReadInts2 does when it fails.

b) Write ReadInts3 that does not fail no matter what characters follow the final integer.

c) Write ReadInts4 to permit arbitrary extraneous characters to follow *any* of the integers in the file. (Warning—the most difficult part of this exercise is in stating the requirements for ReadInts4. Spend as much time as you can thinking about what the program is supposed to do before starting a design.)

14.3.4 Which (if any) of the following expressions are legal? Justify your answers by drawing syntax trees.

a) `--X`

b) `-(-X)`

c) `A * B < C`

d) `X * -Y`

14.3.5 Given execution state {A·5, B·25, C·10}, work out the meanings of the following expressions:

```
B * C MOD (A + 3)
B * C MOD A + 3
B + C DIV A + 3
B + C DIV (A + 3)
B MOD (A DIV B)
```

14.3.6 Given an integer N in the input, design and test a program to find the sum of integers up to N.

14.3.7 Given an integer N in the input, design and test a program to find the sum $1^1 + 2^2 + \cdots + N^N$.

14.3.8 Design and test a procedure to print the value of an `INTEGER` variable, with no preceding blanks. (Hint: compute and use a format size to print the integer.)

14.3.9 Given an integer N in the input, design and test a program to find the sum of all the factors (except 1) of N.

14.3.10 Given an integer N in the input, design and test a program to find the maximum difference between a number and the sum of its factors (except 1) for all numbers between 2 and N. Can you find an analytic solution to this problem?

14.3.11 Given a file of integers, determine the minimum, maximum, and average (including fractional part to two decimal places).

14.3.12 Given a file of integers, determine the median.

14.3.13 Design and test a module for doing extra-precision addition and multiplication for numbers stored in representations of your choice on a file.

14.4 Subrange Types

Preview

When a variable should assume values in a limited range, it may be restricted to a subrange of another ordinal type. The restriction helps to catch mistakes that would send the values out of range.

New Pascal types can also be formed by specifying their values to *lie within a subrange of the* values of an existing ordinal type. The subrange is defined by listing the minimum and maximum constants that are part of the subrange.

<subrange type> ::= <constant> . . <constant>

<constant> ::= <sign> <unsigned number> ¦ <unsigned number>
¦ <constant identifier> ¦ <character string>

Consider the following declarations:

```
TYPE
  Letter = 'A' .. 'Z';
  SmallInt = 0 .. 100;
  SmallerInt = 1 .. 10;
VAR
  L: Letter;
  X: SmallInt;
  Y: SmallerInt;
```

The type from which the constants defining the subrange type are drawn is called the *host type*. All operations of the host type may be applied to operands of the subrange type. In the example above, the host type of `Letter` is `CHAR`, and that of `SmallInt` and `SmallerInt` is `INTEGER`. Thus `CHAR` operations can be applied to `L`, and `INTEGER` operations to `X` or `Y`.

```
BEGIN
  ...
  IF L = 'A' THEN ... {compare Letter and CHAR}
  WRITE(X + Y); {add INTEGER values}
  ...
END
```

Once an enumerated type has been declared, its constants may be used to define subrange types whose host type is the enumerated type. The declaration of new subrange types (like `WeekDay` and `WeekEndDay` in the following example) are not duplicate declarations of the identifiers used to define the subrange type.

```
TYPE
  DayOfWeek = (Monday, Tuesday, Wednesday, Thursday,
               Friday, Saturday, Sunday);
  WeekDay = Monday .. Friday;
  WeekEndDay = Saturday .. Sunday;
```

(Since character strings are constants, the syntax rules seem to permit defining subranges of character strings; however, this is not legal.)

14.4.1 Subrange Assignments

It is not always possible to determine in advance whether a statement containing subrange values can violate the subrange boundaries. Consider the following assignment statement involving `SmallInt X` and `SmallerInt Y` declared above:

```
Y := Y + X
```

Since the host type is `INTEGER`, it is legal to apply the addition operator to these operands. The addition yields an `INTEGER` value that may be assigned to `Y` because its host type is `INTEGER`. However, if the result of addition is outside the declared range of values for `Y` (i.e., `0..10`), program execution will terminate with an error message.

Subrange types require redefinition of the meaning of the assignment statement so that the function of the assignment statement (but not that of the expression) is undefined if the value of the expression is outside the declared range of the variable on the left side of the assignment.

$$\boxed{V := E} = \{<s, t>: \boxed{E}(s) \in Vals(V),\ t = (s - \{<V, c>: c \in Vals(V)\}) \cup \{<V, \boxed{E}(s)>\}\},$$

where $Vals(V)$ is the set of all values for the type of V.

14.4.2 Temperature Conversion

The following example converts temperatures below the boiling point of water from Farenheit to Celsius. Both variables could be declared as integers; however, knowledge of the problem domain provides a meaningful restriction of the values.

```
PROGRAM Temperature(INPUT,OUTPUT);
  VAR
    Farenheit: -459 .. 212;
    Celsius: -273 .. 100;
  BEGIN
    READ(Farenheit);
    Celsius := (Farenheit - 32)*5 DIV 9;
    WRITELN(Farenheit:4,' degrees Farenheit or',
            Celsius:4,' degrees Celsius')
  END.
```

Execution

```
INPUT :212
OUTPUT: 212 degrees Farenheit or 100 degrees Celsius
INPUT :32
OUTPUT:  32 degrees Farenheit or   0 degrees Celsius
INPUT :-459
OUTPUT:-459 degrees Farenheit or-272 degrees Celsius
INPUT :1000
OUTPUT:Value 1000 is out of range
```

The last output is not produced by the program but by the Pascal machine which detects that 1000 is outside the range of allowable values for Farenheit.

Subrange types help document a program and shift the burden of checking the legality of assignments from the programmer to the Pascal machine. This often results in a smaller, more easily understood program. In addition, values with subrange types may require less storage than identical host-type values.

14.4.3 Exercises

14.4.1 Which of the following are correct subrange types:

a) `Letter = 0 .. 100`

b) `Reverse = 'Z' .. 'A'`

c) `Temp = 32 ... 212`

d) `One = 1 .. 1`

14.4.2 The use of subrange types in assignment statements slows down a Pascal program when checks are made.

a) Explain why the speed suffers only when the subrange variable appears on the left side of the assignment.

b) Make timing comparisons on your computer to determine how bad the slowdown is.

14.4.3 Suppose that for a particular computer, *MAXINT* is 1,000,000. Explain why a programmer might want to declare:

```
VAR
  X: -1000000 .. 1000000
```

instead of

```
VAR
  X: INTEGER
```

14.5 Design and Analysis with Ordinal Types

Preview

The design rules for CF Pascal extend to ordinal types with little change. Analysis of programs is also similar, but the operators of the INTEGER type require some new techniques. The accumulating iteration is particularly easy to analyze.

14.5.1 Design Rules

The design rules for BEGIN, IF, and WHILE statements in CF Pascal can be extended for D Pascal and its ordinal types. The single change required is in the preservation rule for BEGIN statements. For CF Pascal it is:

> At each step, preserve all original values of the right side of the intended concurrent assignment.

The word "preserve" can be replaced by the phrase "preserve the ability to compute," that is:

> At each step, preserve the ability to compute all original values of the right side of the intended concurrent assignment.

Since arithmetic operations have inverses, satisfying the new preservation rule will not always require introducing temporary variables.

Consider the problem of designing a BEGIN statement to exchange two integer values without using a temporary variable.

```
X, Y := Y, X
```

Although a temporary variable would be required to preserve either the value of X or Y before it is changed by the first assignment, one value and the sum of the two values preserves the ability to compute them both, since the value of one variable can be subtracted from the sum to obtain the other. That is, the following design is acceptable:

```
BEGIN
  X := X+Y;
  Y := X-Y;
  X := X-Y
END
```

The values of X and Y must of course lie in the interval [-*MAXINT*, *MAXINT*], but the interchange will not operate as desired for all such values. The results of each operation must also fall within this interval. The conditions of correct execution are calculated in the following symbolic execution. The abbreviation *inrange*(X) means that the value of X is within the interval [-*MAXINT*, *MAXINT*].

Part	Condition	X	Y
		X	Y
X := X + Y	*inrange*(X+Y)	X+Y	
Y := X - Y	*inrange*(X+Y-Y)		X+Y-Y=X
X := X - Y	*inrange*(X+Y-X)	X+Y-X=Y	

The conditions evidently all hold if the first one does, and this also forces *inrange*(X) and *inrange*(Y), so the function computed by this exchange code is actually:

(*inrange* (X+Y) → X,Y := Y,X)

Overflow is not often treated this formally, but the problem always exists.

As another example of design using the INTEGER type, consider simulating addition by repeatedly adding one. That is, to add the values of X and Y, design a WHILE statement to add one to X, Y times. Overflow will henceforth be ignored, so the desired function is:

f = (0 <= Y → X := X+Y)

The existence condition of the WHILE design rule requires that range(f) $\subseteq$ domain(f), which is satisfied; and, for each $s \in$ range(f), $f(s) = s$. The latter condition fails. Consider a state s in range(f) for which the value of Y is nonzero. f maps this state to one in which the value of X is replaced by the value of X+Y, which is necessarily different, hence the new state cannot be s as required by the existence condition.

Perhaps the WHILE statement could be designed if the value of Y is allowed to change, but the same argument shows that the second part of the existence condition will fail unless the new value of Y is zero. Thus consider

g = (0 <= Y → X,Y := X+Y,0)

as the desired function. The range(g) is those states in which the value of Y is zero, and g is defined on all such states, so

range(g) $\subseteq$ domain(g)

is satisfied. So is the second part of the existence condition,

$g(s) = s$ for all $s \in$ range(g),

because any state $s \in$ range(g) has Y value zero, and so g adds zero to the X value, leaving it unchanged.

With the existence condition satisfied, the WHILE condition B must be chosen such that:

$\boxed{B}(s)$ is TRUE for all $s \in$ domain(g) - range(g),

$\boxed{B}(s)$ is FALSE for all $s \in$ range(g).

In domain(g), the value of Y is nonnegative, and since the Y value is zero in range(g), the simplest expression for B is Y>0.

Finally, a BEGIN statement D must be chosen such that

WHILE Y>0 DO D

terminates for every state in domain(g), and

$\boxed{\text{IF Y>0 THEN } D}$

preserves the values needed to reach the final state, that is, X+Y.

To make progress toward termination Y can be decremented. In order to preserve X+Y, X must be incremented by the same amount that Y is decremented. Thus the WHILE statement designed for g is:

```
WHILE Y > 0
DO
  BEGIN
    Y := Y - 1;
    X := X + 1
  END
```

14.5.2 Program Analysis

Programming tasks do not always begin with design—often an existing program must be understood in order to modify it, for example. Because the program analysis techniques developed for CF Pascal depend only on the definition of the box functions, they can be applied to additional data types like INTEGER. Consider the program fragment:

```
BEGIN
  X := 3;
  Y := 5;
  {(Y>=0 --> X,Y := X+Y,0) | (Y<0  -->  )}
  WHILE Y > 0
  DO
    BEGIN
      Y := Y - 1;
      X := X + 1
    END
END
```

The intended function F for the WHILE statement is given by the comment:

F = (Y>=0 --> X,Y := X+Y,0) | (Y<0 -->)

The WHILE verification rule requires the following three conditions:

1. domain(F) = domain(`WHILE Y>0 DO BEGIN Y:=Y-1; X:=X+1 END`)
2. (Y<=0 → F) = (Y<=0 →)
3. F = `IF Y>0 THEN BEGIN Y:=Y-1; X:=X+1 END` ∘ F

The domain of F is all states, so termination is required for all values of Y. If the value of Y is negative or zero, the WHILE statement is skipped so termination is assured. If the value of Y is positive, the WHILE statement is executed and Y is decremented, so eventual termination is assured because the value of Y approaches 0. Thus the first condition is satisfied.

We can rewrite the left side of the second condition so that it is identical to the right side.

```
Y<=0 → F
= (Y<=0 → (Y>=0 → X,Y := X+Y,0)) | (Y<=0 → (Y<0 →  ))
= (Y<=0 AND Y>=0 → X,Y := X+Y,0) | (Y<=0 AND Y<0 →  )
= (Y=0 → X,Y := X+Y,0) | (Y<0 →  )
= (Y<=0 →  )
```

Finally, consider the right side of the third condition:

`IF Y>0 THEN BEGIN Y:=Y-1; X:=X+1 END` ∘ F

The function corresponding to the IF statement is:

`IF Y>0 THEN...` = (Y>0 → X,Y := X+1,Y-1) | (Y<=0 →)

so `IF Y>0 THEN...` ∘ F is:

```
((Y>0 → X,Y := X+1,Y-1) | (Y<=0 →  )) ∘
   (Y>=0 → X,Y := X+Y,0) | (Y<0 →  )
```

There are four cases to consider, corresponding to the first and second conditions of the two conditional assignments. Name these cases by which conditions are selected; for example, case 1-2 means the first condition of `IF Y>0 THEN...` and the second condition of F.

Case 1-1

Part	Condition	X	Y
IF ...	Y>0	X + 1	Y - 1
F	Y-1>=0	X+1+Y-1 = X+Y	0

The condition simplifies to Y>=1, so the function is:

(Y>=1 → X,Y := X+Y,0)

Case 1-2

Part	Condition	X	Y
IF ...	Y>0	X + 1	Y - 1
F	Y-1<0		

The condition is:

Y>0 AND Y-1<0 = Y>0 AND Y<1

which cannot be satisfied, so this case does not occur.

Case 2-1

Part	Condition	X	Y
IF ...	Y<=0		
F	Y>=0	X+Y	0

The condition simplifies to Y=0, so the function is

(Y=0 →)

because when Y is zero, setting it to zero and adding it to X makes no change.

Case 2-2

Part	Condition	X	Y
IF ...	Y<=0		
F	Y<0		

Thus the function is:

(Y<0 →)

Putting the four part functions together:

```
(Y>=1 → X,Y := X+Y,0) | (Y=0 → ) | (Y<0 → )
= (Y>0 → X,Y := X+Y,0) | (Y<=0 → )
```

Thus the right side of the third condition is identical to *F*, as was to be shown.

Since the conditions of the WHILE verification rule are satisfied, *F* is the function computed by the WHILE statement.

Composing *F* with the function of the first assignment statements:

(X,Y := 3,5) ∘ *F*

gives the function for the entire program fragment:

```
(5>0 → X,Y := 3+5,0) | (5<=0 → X,Y := 3,5)
= (X,Y := 8,0)
```

The intended function for a WHILE statement is not always given as a conditional assignment; often the intended function must be obtained from the program. In the previous example, it would have been easy to determine the function for the WHILE statement because both the assignment statements in it are *accumulating* assignment statements. In an accumulating assignment a variable's value is altered by adding or subtracting a fixed value. The mathematical concept of summation (i.e., repeated addition) is a natural choice for a function to describe the repeated effects on X and Y. Since the loop is executed as long as Y>0, and the value of Y is decreased by 1 during each execution, the summation will contain Y terms. On each iteration, since both X and Y are altered by the constant 1, this is the term to be summed. The sum is added to the initial value of X and subtracted from the initial value of Y. Thus the WHILE-statement function is:

$$X,Y := X + \sum_{k=1}^{Y} 1,\ Y - \sum_{k=1}^{Y} 1$$

These values can be simplified:

$$X + \sum_{k=1}^{Y} 1 = X + 1(Y-1+1) = X+Y-1+1 = X+Y$$

$$Y - \sum_{k=1}^{Y} 1 = Y - 1(Y-1+1) = Y-Y-1+1 = 0$$

so the function is:

```
(X,Y := X+Y,0)
```

as before.

This technique can be used to determine the function of another WHILE statement. The Pascal fragment below was written to simulate the operators DIV and MOD using only addition and subtraction. Its function is intended to be:

```
f = (Numerator>=0 AND Denominator>0 →
      Quotient, Remainder :=
      Numerator DIV Denominator, Numerator MOD Denominator)
```

While the function of the entire fragment is given, the function computed by the WHILE statement must be determined in order to verify that the fragment computes f.

```
Quotient := 0;
Remainder := Numerator;
WHILE Remainder >= Denominator
DO
  BEGIN
    Quotient := Quotient + 1;
    Remainder := Remainder - Denominator
  END
```

Let the values of Quotient, Remainder, and Denominator be Q, R, and D, respectively. The test for termination is $R \geq D$, or $R - D \geq 0$, so the loop will be executed k times, where

$$k = (R\text{-}D) \div D = R \div D - 1$$

Thus, Q and R will be changed to:

$$Q + \sum_{k=1}^{R \div D} 1$$

$$R - \sum_{k=1}^{R \div D} D$$

respectively. These expressions reduce to:

$$Q + \sum_{k=1}^{R \div D} 1 = Q + 1(R \div D)$$

$$R - \sum_{k=1}^{R \div D} D = R - D(R \div D).$$

The function for the WHILE statement is thus:

```
(Remainder>=Denominator AND Denominator>0 →
   Remainder,Quotient :=
      Remainder - (Remainder DIV Denominator) * Denominator,
      Quotient + (Remainder DIV Denominator)) |
(Remainder<Denominator →  )
```

The verification that this is the WHILE-statement function, and that the program fragment is correct for the given intended function *f*, are similar to those in the previous example.

14.5.3 Exercises

14.5.1 Given the problem of designing a BEGIN statement to exchange two integer values without using a temporary variable, i.e., to achieve

```
X,Y := Y,X
```

the first step

```
BEGIN
  X := X - Y;
  ...
END
```

satisfies the preservation rule. Complete the design of this BEGIN statement and determine the actual condition for its validity because INTEGER values are restricted to the range [-*MAXINT*, *MAXINT*].

14.5.2 Use the design rule for BEGIN statements to achieve

```
X,Y,Z := X-Y,X+Y,X
```

14.5.3 Show that the function:

$$(Y>0 \rightarrow X,Y := X - \sum_{i=1}^{Y} Y - i,\ 0\ )\ |\ (Y \leq 0 \rightarrow\ )$$

is computed by the WHILE statement:

```
WHILE Y > 0
DO
  BEGIN
    Y := Y - 1;
    X := X - Y
  END
```

14.5.4 Show that the function:

```
(X>=0 → X := X MOD 3)
```

is computed by the WHILE statement:

```
WHILE (X>2) OR (X<0)
DO
  X := X - 3
```

14.5.5 Show that the function:

```
(X>Y → X,Y := X-2((X-Y+3) DIV 4), Y+2((X-Y+3) DIV 4)) |
(X<=Y →  )
```

is computed by the WHILE statement:

```
WHILE X > Y
DO
  BEGIN
    X := X - 2;
    Y := Y + 2
  END
```

14.5.6 The Fibonacci numbers are defined recursively as:

$$Fib(n) \equiv \begin{cases} Fib(n-1) + Fib(n-2) & \text{if } n \geq 2 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}.$$

Consider the following WHILE statement:

```
WHILE (K<N) AND (N>0)
DO
  BEGIN
    K := K + 1;
    F := F + G;
    G := F - G
  END
```

Instead of determining the function for all the variables in the WHILE statement, restrict attention to F and demonstrate

```
WHILE (K<N) ... | =

  (K<N AND N>0 → F := Fib(N-K+1)*F + Fib(N-K)*G) |
  (K>=N OR N<=0 → )
```

14.5.7 Consider the following procedure:

```
PROCEDURE Fact(VAR M,A: INTEGER);
  BEGIN {Fact}
    IF A > 1
    THEN
      BEGIN
        A := A - 1;
        Fact(M,A);
        A := A + 1;
        M := M * A
      END
    ELSE
      M := 1
  END  {Fact}
```

Show that

$$\boxed{\texttt{Fact(M,A)}}(\{\texttt{M}\cdot ?,\ \texttt{A}\cdot a\}) = \{\texttt{M}\cdot a!,\ \texttt{A}\cdot a\}$$

if $a \geq 0$. What is the result if $a < 0$?

14.5.8 Is the following recursive procedure:

```
PROCEDURE Sum(VAR L,H,I,R: INTEGER);
  BEGIN {Sum}
    IF L <= H
    THEN
      IF L = H
      THEN
        R := I + R
      ELSE
        BEGIN
          R := R + I;
          L := L + 1;
          Sum(L,H,I,R);
          L := L - 1
        END
    ELSE
      R := 29
  END  {Sum}
```

correct with respect to the specification:

```
(L<=H → R := R+I*(H-L+1))
```

14.5.9 What functions are computed by the following Pascal fragments?

a)
```
WHILE T <> Y
DO
  BEGIN
    W := W * X; T := T + 1
  END
```

b)
```
WHILE X <> N
DO
  BEGIN
    X := X + 3; Y := Y + X
  END
```

c)
```
S := 0;
WHILE X <> 0
DO
  BEGIN
    T := 0;
    WHILE T <> Y
    DO
      BEGIN
        S := S + 1;
        T := T + 1
      END;
    X := X - 1
  END
```

14.5.10 What function is computed by the procedure statement:

```
Sum(X,Y,Z)
```

where Sum is declared as follows:

```
PROCEDURE Sum(VAR X,Y,Z: INTEGER);
  BEGIN {Sum}
    IF X <= Y
    THEN
      BEGIN
        X := X + 1;
        Z := Z + X
      END
    ELSE
      BEGIN
        Y := Y + 1;
        Z := Z + Y;
        Sum(X,Y,Z)
      END
  END  {Sum}
```

14.5.11 Write a Pascal fragment to implement the function:

```
Arb,Extra := Arb+Extra,0
```

Verify your answer.

14.5.12 Write a Pascal fragment to implement the function:

(X<>0 AND Y>0 → X,Y := X^{2^Y},0)

Verify your answer.

14.5.13 For each of the following functions, either design a WHILE statement to compute the function or explain why it cannot be done.

a) $f(X) = X \ mod \ 10$

b) $f(X) = X^2$

c) $f(X) = \begin{cases} X \text{ if } X < 100 \\ 100 \text{ if } X \geq 100 \end{cases}$

d) $f(X) = \begin{cases} 0 \text{ if } X \textit{ is odd} \\ 1 \text{ if } X \textit{ is even} \end{cases}$

e) $f(X) = \begin{cases} 0 \text{ if } X = 0 \\ 1 \text{ if } X = 1 \\ 2 \text{ if } X \textit{ is prime} \\ 3 \text{ if } X \textit{ is not prime and } X \geq 2 \end{cases}$

f) $f(X,Y) = \begin{cases} (X,X+1) \text{ if } X > 100 \\ (X,Y) \text{ if } X \leq 100 \end{cases}$

g) $f(X,Y) = \begin{cases} (X,Y+1) \text{ if } X > 100 \\ (X,Y) \text{ if } X \leq 100 \end{cases}$

14.6 Summary

In this chapter, Pascal's ordinal data types have been examined. The operations that can be applied to values of each type are summarized below.

Operator	Operation	Functionality
NOT	negation	Boolean → Boolean
AND	conjunction	Boolean × Boolean → Boolean
OR	disjunction	
=	equality	ordinal × ordinal → Boolean
<>	inequality	(ordinal types must match,
<	less	subranges are considered to
<=	less or equal	have the host type)
>	greater	
>=	greater or equal	
+	unary +	Integer → Integer
-	unary -	
+	binary addition	Integer × Integer → Integer
-	binary subtraction	
*	multiplication	
DIV	integer division	
MOD	modulus	

Each ordinal type has its use. BOOLEAN variables can be used to record complex conditions for later testing. INTEGER variables make counting easy, so long as the restriction to [-*MAXINT*, *MAXINT*] is observed. Enumerated types are good for capturing a small set of values, each with a mnemonic name. Subrange types allow the programmer to declare value bounds, that are then checked automatically.

The analysis methods used for CF Pascal extend to the the ordinal data types almost without change.

CHAPTER 15

THE PROGRAM STRUCTURE OF D PASCAL

Chapter Preview

D Pascal programs may include sections defining constants and types for use within the program. They may also define and use functions in much the same way as procedures. They can handle files that exist outside the program, perhaps created by some other means, or to be passed on for another use. These features are a convenience, but they also contribute to good programming style by documenting and localizing program meaning.

CF Pascal programs consist of a heading, variable declarations, procedure declarations, and a body. D Pascal is a richer language than CF Pascal, containing additional features such as external files, constants, types, and functions, and its program structure reflects this. The <program heading> and <block> of a D Pascal program may contain more features than in a CF Pascal program. The syntax for D Pascal is:

<program> ::= <program heading> ; <block> .

<program heading> ::= PROGRAM <identifier> (<identifier list>)

<identifier list> ::= <identifier list> , <identifier>

<block> ::= <constant definition part> <type definition part>
<variable declaration part> <procedure and function declaration part>
<BEGIN statement>

<variable declaration part> ::= VAR <variable declarations> ; ¦

<variable declarations> ::= <variable declarations> <variable declaration>
¦ <variable declaration>

<variable declaration> ::= <identifier list> : <type denoter>

The order of these parts is very rigid, but is easy to remember from Pascal's requirement that identifiers be declared before use. Constant identifiers, which are declared in the constant definition part and may be used in subrange declarations in the type definition part, are defined before any types. Type identifiers, which are declared in the type definition part and used in variable declarations in the variable declaration part, are defined before any variables. Similarly, variable declarations must appear before the uses of variables in procedures and procedures must be declared before they are called from the program's BEGIN statement. External files are an exception because they appear in the program heading, yet are later declared in the program's block.

15.1 External Files

Preview

Files whose existence goes beyond the execution time of programs can be used to create program systems, and to communicate between Pascal programs and other parts of a computer system.

Files may have lifetimes (or extents) like those of other Pascal values (limited to the duration of procedure or program execution), but they may also continue to exist outside program execution. If a file variable appears in a program heading (with `INPUT` and `OUTPUT`), the file is called *external*. The contents of an external file may be read or written by a Pascal program, but the file may exist before execution begins, and it remains after program execution terminates. For example, in the

program

```
PROGRAM FileProcess(INPUT,OUTPUT,ExtFile);
  VAR
    IntFile, ExtFile: TEXT;
  BEGIN
    ...
  END.
```

the contents of ExtFile may exist before FileProcess executes, and remains after execution, but the contents of IntFile must be created within FileProcess, and will be destroyed at the end of execution. Each Pascal machine has different ways of associating Pascal external files with names in its file system. Often the Pascal identifier is the name used, but may be subject to restrictions outside Pascal.

External files can have three different uses: input (supplying data to programs from outside), output (holding data supplied by this program for other programs), and update (bringing data to the program to be updated for later use). In illustration, an external file could be used to maintain a cumulative record of students preregistered in a course. Suppose the standard input file INPUT contains the names of all students who registered today, and Students contains the names of previously registered students in sorted order. The following design part shows the processing required.

```
PROGRAM Register(INPUT, OUTPUT, Students);
  {add names in INPUT to those in Students and
   report any duplicate names in OUTPUT}
  VAR
    Students, Temp1, Temp2: TEXT;
  BEGIN {Register}
    {copy INPUT to Temp1};
    {sort names in Temp1};
    {merge Temp1 and Students into Temp2--
     if a name occurs in both, send to OUTPUT
     and don't duplicate in Temp2};
    {copy Temp2 to Students}
  END.  {Register}
```

Temp1 and Temp2 are internal files that disappear at the end of execution. Students has been updated and will be used in the same program tomorrow (with tomorrow's INPUT).

With external files D Pascal provides the capability of creating *program systems*, not simply single programs, in which programs exchange information through external files. For example, in the illustration above, another program might print the file Students, say as in the design part:

```
PROGRAM PrintRegistered(INPUT, OUTPUT, Students);
  {print the names in Students to OUTPUT}
  VAR
    Students: TEXT;
  BEGIN {PrintRegistered}
    {copy and format names from Student to OUTPUT}
  END.  {PrintRegistered}
```

Program systems require careful design just as do programs, but now the data structures are the external files, and the elementary operations are the actions of the programs within the system. In data processing applications, the person who designs program systems, but not necessarily the programs within the system, is called a *system analyst*.

15.1.1 Exercises

15.1.1 Find out what name is actually entered in the system directory when a Pascal program writes an external file named `MyFile`.

15.1.2 Find out what happens on your Pascal machine if you try to read an external file that does not in fact exist before your program begins execution. Can this information be used to write a test for a file that does not yet exist (so that, for example, the program could then create it)?

15.1.3 Write a trivial program system consisting of three programs as follows:

Program A creates an empty external file.

Program B updates the external file by adding to it a line copied from the input.

Program C prints out the contents of the external file.

Write a "users' manual" for this program system that tells how to use it as a one-line-per-day diary.

15.2 Constants

Preview

By naming constant values, the programmer can change them throughout a program without having to search for each occurrence.

Constant values can be given names using declarations like the following:

```
CONST
  LastCh = '#';
  FieldWidth = 4;
  Max = 9999;
  Min = -Max;
  Name = 'Blaise Pascal';
```

Since a constant identifier introduced by a constant declaration names a value, it has the same type as the value and can appear anywhere the value could. In the following example based on the above `CONST` declarations, `Min` and `Max` are used to declare a subrange type; `LastCh` is compared to a character value; `FieldWidth` controls the size of the field in which the `SmallInt` values are printed; and `Name` is used in a WRITE statement.

```
TYPE
  SmallInt = Min ..  Max;
VAR
  Value: SmallInt;
  Ch: CHAR;
BEGIN
  ...
  WHILE Ch <> LastCh
  DO
    ...
  READ(Value);
  WRITELN(Value:FieldWidth);
  WRITELN(Name);
  ...
```

The syntax for constant declarations appears below.

<constant definition part> ::= CONST <constant definitions> |

<constant definitions> ::= <constant definitions> <constant definition>
| <constant definition>

<constant definition> ::= <identifier> = <constant> ;

<constant> ::= <sign> <unsigned number> | <unsigned number>
| <sign> <constant identifier> | <constant identifier>
| <character string>

<constant identifier> ::= <identifier>

The context rules that accompany this syntax state that only the <identifier>s that occur in <constant definition>s are <constant identifier>s. These special identifiers may not be used as if they were variables, because their values are fixed. For example, a constant identifier cannot appear on the left side of an assignment statement or as an actual parameter corresponding to a formal VAR parameter.

Constant identifiers help document programs and make program changes easier. Declaring the constant `LastCh` with fixed value `#`, and using `LastCh` instead of the literal `'#'` improves a program. Consider the following fragment:

```
CONST
  LastCh = '#';
VAR
  Ch: CHAR;
BEGIN
  ...
  WHILE Ch <> LastCh
  DO
    ...;
  WRITELN('Inventory # is:', ...)
END
```

If we decided to end files with another character (say `$`) in order to process a file that contained the character `#`, only the constant declaration in the program fragment need be changed:

```
CONST
  LastCh = '$';
```

Searching through the entire program changing uses of `#` to `$` is both tedious and error-prone. For example, the `#` in the final WRITE statement might be changed by mistake.

15.2.1 Exercises

15.2.1 Which of the following are legal constant declarations in Pascal?

```
CONST
  CharsPerWord: 4;
  StringLength = 15;
  WordsPerString = (StrLength+CharPerWord-1) DIV CharPerWord;
  LowInt = -MAXINT;
  Hello = 'Hello';
```

15.2.2 Explain why it is reasonable to forbid passing a constant identifier as an actual VAR parameter to a procedure.

15.2.3 Give an example in which the same constant identifier is used in both the <variable declaration part> and the <BEGIN statement> of a program so as to prevent a potential inconsistency between declaration and usage.

15.3 Types

Preview

Because types are an important feature of Pascal, it is advantageous to be able to name them, and use the name instead of the full type description when needed. By defining a complex type only once, the programmer can prevent accidental transcription errors and make it easy to alter every occurrence of that type.

Although data types describe sets of values and the operations that can be applied to the values, Pascal type declarations only describe the representation of the values. New types are declared in the <type definition part> section of a block by associating an identifier (the name of the new type) with a description of its values (e.g., listing the constants of an enumerated type or the bounds of a subrange type). Some examples of new types appear below.

```
TYPE
  CType = CHAR;
  Ct = CType;
  Month = (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);
  Summer = Jun ..  Aug;
  DayNum = 1 .. 31;
  Letter = 'A' .. 'Z';
```

Type identifiers like `Letter` can occur anywhere primitive types occur in declarations and formal parameter lists. Type identifiers are a convenient shorthand—by naming a list of constant identifiers like `Month` it can be easily repeated in a program without the danger that it might be copied incorrectly.

The syntax of type declarations appears below:

<type definition part> ::= TYPE <type definitions> ¦

<type definitions> ::= <type definitions> <type definition>
¦ <type definition>

<type definition> ::= <identifier> = <type denoter> ;

<type denoter> ::= <type identifier> ¦ <new type>

<new type> ::= <new ordinal type>

<new ordinal type> ::= <enumerated type> ¦ <subrange type>

The most important use of type identifiers is in formal parameter lists.

<formal parameter list> ::= (<formal parameters>) ¦

<formal parameters> ::= <formal parameters> ; <formal parameter>
¦ <formal parameter>

<formal parameter> ::= <variable parameter>

<variable parameter> ::= VAR <identifier list> : <type identifier>

Each formal parameter *must* be declared with a type identifier. Thus none of the formal parameters in the following procedure heading are legally declared:

```
PROCEDURE Proc
  (VAR Mo: (NoMonth, Jan, Feb, Mar, Apr, May,
            Jun, Jul, Aug, Sep, Oct, Nov, Dec);
   VAR Day: 1 .. 31;
   VAR FInitial, LInitial: 'A' .. 'Z');
```

because they are not declared with type identifiers. By introducing type declarations, the procedure heading can be correctly written as follows:

```
TYPE
  Month = (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);
  DayNum = 1 .. 31;
  Letter = 'A' .. 'Z';

PROCEDURE Proc(VAR Mo: Month; VAR Day: DayNum;
               VAR FInitial, LInitial: Letter);
```

The ability to define new types raises the question of *type equality*: when are two types the same? Many of the operations in programs depend on their operands having matching types, and Pascal has three degrees of type compatibility to handle different situations: *same, assignment compatible,* and *compatible.* The context rules governing type agreement are:

1. Actual parameters must have the *same* types as their corresponding variable formal parameters.
2. The type of the variable on the left side of an assignment statement must be *assignment compatible* with the type of expression on the right side.
3. The operands of relational operators must have *compatible* types.

In Pascal, two types are the *same* if they have been defined as being equivalent in a type definition. Variables have the same type if they are declared with type identifiers that are the same or if they are declared together in a variable declaration. For example, consider the declarations:

```
TYPE
  T1 = INTEGER;
  T2 = T1;
  SmallInt = 0 .. 9;
  Age = 0 .. 9;
VAR
  W: SmallInt;
  X: 0 .. 9;
  Y, Z: 0 .. 9;
```

The types `INTEGER`, `T1`, and `T2` are all the same in the above example. `SmallInt` and `Age` are not the same types. `Y` and `Z` have the same type, but technically `X` has a different type. In the following Pascal fragment, using the above declarations, only `W` may be passed to `P` since it is the only variable with the same type as `Parm`.

```
PROCEDURE P(VAR Parm: SmallInt);
  ...
  P(W); {legal -- Parm and W have type SmallInt}
  P(X); {not legal -- Parm and X have different types}
  P(Y); {not legal -- Parm and Y have different types}
  ...
```

Two types, `T1` and `T2`, are *compatible* if either of the following is true:

`T1` and `T2` are the same type.

`T1` is a subrange of `T2`, or `T2` is a subrange of `T1`, or both `T1` and `T2` are subranges of the same host type.

In the above example, the type of X is compatible with Y because both are subranges of INTEGER. The variables Letter and Digit in the following example have types that are compatible because they are both subranges of the host type CHAR.

```
VAR
  Letter: 'A' .. 'Z';
  Digit: '0' .. '9';
BEGIN
  ...
  IF (Digit <= '0') OR (Digit = Letter)
  ...
```

Thus Digit can be compared to the constant '0' (which has type CHAR) and to Letter.

A value of type T2 is *assignment-compatible* with a type T1 if either of the following is true:

T1 and T2 are the same type (but not a file type).

T1 and T2 are compatible ordinal types and the value of type T2 is in the closed interval specified by the values of type T1.

In the following example, only the first two assignment statements are legal.

```
TYPE
  SmallInt = 0 .. 99;
  SmallerInt = 0 .. 9;
VAR
  X: SmallInt;
  Y: SmallerInt;
BEGIN
  X := 0;  {The type of X (SmallInt) is compatible with
            the type of 0 (INTEGER) and 0 is in
            SmallInt's closed interval}
  Y := X;  {The type of Y (SmallerInt) is compatible
            with the type of X (SmallInt) and the value
            of X (0) is in SmallerInt's
            closed interval}
  Y := X-1 {The type of Y (SmallerInt) is compatible
            with the type of X-1 (INTEGER) but the
            value of X-1 (-1) is outside SmallerInt's
            closed interval}
END
```

15.3.1 Exercises

15.3.1 In the following Pascal fragment, identify the errors (if any).

```
TYPE
  SmallInt = 0 .. 99;
  SmallerInt = 0 .. 9;
VAR
  W: 0 .. 9;
  X: SmallInt;
  Y: SmallerInt;
  Z: 0 .. 9;
  Letter: 'A' .. 'Z';
  Digit: '0' .. '9';
```

```
PROCEDURE P(VAR Parm1: SmallInt; VAR Parm2: SmallerInt);
  BEGIN {P}
    Parm1 := Parm2
  END;  {P}
BEGIN
  Digit := '0';
  Letter := 'A';
  Z := 0;
  X := 0;
  W := Z;
  IF (Digit <= '0') OR (Digit = Letter)
  THEN
    WRITELN('hello');
  Y := X;
  P(X,Z);
  P(X,X-1);
  P(W,Y)
END
```

15.3.2 The technical rules about Pascal type agreement for variable usage range from obviously necessary to rather arbitrary. Give an example of a rule that is obviously necessary and explain why. Give an example of a rule that seems merely arbitrary.

15.4 Value and Variable Parameters

Preview

Parameters passed by value are an isolation mechanism between the point of call and the statements of the called routine. No actions in the routine can transmit information back to the caller through the value parameter. Thus these parameters serve as a safe way to provide read-only input values.

There are two kinds of formal parameters in D Pascal: *variable* parameters and *value* parameters. In a procedure declaration variable parameters look like declaration statements. They begin with the reserved word VAR, followed by a list of identifiers separated by commas that are the names of the formal parameters, a colon, and an identifier giving the parameter type. Value parameters are declared like variable parameters except that the leading VAR is omitted. In the following example, VarParm is a variable formal parameter of type T and ValParm is a value formal parameter of type T.

```
PROCEDURE P(VAR VarParm: T; ValParm: T);
```

The syntax rules describing parameters are:

<formal parameter> ::= <variable parameter> | <value parameter>

<variable parameter> ::= VAR <identifier list> : <type identifier>

<value parameter> ::= <identifier list> : <type identifier>

When an actual parameter is bound to a variable formal parameter, an *alias* is set up between the name of the actual parameter (which must be a variable rather than a constant or expression) and the name of the formal parameter. The word alias is used because when the procedure statement is replaced by the body of its procedure declaration, variable formal parameters are replaced by their corresponding actual parameters. Thus, each time a value is assigned to a variable formal parameter the value of the aliased actual parameter is changed. The type of the actual parameter must be the *same* as the type of the formal parameter.

Many parameters are used as inputs only; no value is returned through these parameters. Input-only parameters should be declared as value formal parameters to distinguish them from variable formal parameters, which are used as output or input-output parameters. When an actual parameter is bound to a value formal parameter, the value of the actual parameter is assigned to the formal parameter but no alias is established between the formal parameter and the actual parameter. Thus, any assignment to a value formal parameter in a procedure's block changes the value of the formal parameter but not that of the actual parameter. Since the value of the actual parameter is assigned to the value parameter rather than an alias established between the actual and the formal parameters, any expression (including constants, variables, and more complex expressions) may be used as the actual parameter as long as the type of the actual parameter value is *assignment-compatible* with the type of the formal. The difference between variable and value parameters can be seen in VarVsValue.

```
PROGRAM VarVsValue(OUTPUT);
  VAR
    A, B: INTEGER;
  PROCEDURE Bump(VAR VarF: INTEGER; ValF: INTEGER);
    BEGIN
      VarF := VarF + 1;
      ValF := ValF + 1;
      WRITELN(VarF, ValF)
    END;
  BEGIN {VarVsValue}
    A := 0;
    B := 0;
    Bump(A, B);
    WRITELN(A, B)
  END.  {VarVsValue}
```

Execution

```
OUTPUT:          1          1
                 1          0
```

Bump adds 1 to each of its formal parameters so that both VarF and ValF have the value 1 at the end of the procedure. However, the changes made to the formal parameters in Bump appear only in the actual parameter that was bound to the variable formal parameter (A has the value 1 at the end of the program's block.) and not in the actual parameter that was bound to the value formal parameter. (B retained the value it had before the execution of the procedure.)

The meaning for a procedure invocation with variable parameters presented in Chapter 9 can be extended to work for value parameters. Formal value parameters are added and removed from the state just as the procedure's local variables are. After a formal value parameter is added to the state, the new variable is assigned the current value of the actual parameter. With the declaration:

```
PROCEDURE P(Y: Typ);
   T;
```

the body T is added to the state as the value of identifier P. Then the meaning of a procedure statement

```
P( E )
```

is

$$\boxed{\texttt{P(} E \texttt{)}}(s) = \boxed{\texttt{VAR Y: Typ}} \circ \boxed{\texttt{Y := } E} \circ \boxed{s(\texttt{P})} \circ \boxed{\texttt{VAR Y: Typ}}^{\mathrm{T}}.$$

If there are VAR parameters, or confusion between local and global identifiers, these must be modified in s(P) as described in Sections 9.1.4 and 9.2.5, respectively.

This definition can be used to work out the meaning of the call on Bump in VarVsValue. The beginning state is

{OUTPUT·<††,††,W>}.

Then the value of the program function is

```
[VAR A, B: INTEGER] ∘
  [BEGIN A:=0; B:=0; Bump (A, B) ; WRITELN (A, B) END] ∘
    [VAR A, B: INTEGER]^T ∘ [.] ({OUTPUT·<††,††,W>})
= [BEGIN A:=0; B:=0; Bump (A, B) ; WRITELN (A, B) END] ∘
  [VAR A, B: INTEGER]^T ∘ [.] ({OUTPUT·<††,††,W>, A·?, B·?})
= [Bump (A, B)] ∘ [WRITELN (A, B)] ∘ [VAR A, B: INTEGER]^T ∘ [.]
    ({OUTPUT·<††,††,W>, A·0, B·0}).
```

The procedure statement is expanded into a series of function compositions that apply the function corresponding to the declaration of ValF, the assignment of the value of B to ValF, the body of Bump with A substituted for VarF, and the transpose of the declaration of ValF. Thus the above becomes:

```
[VAR ValF: INTEGER] ∘ [ValF := B] ∘ [A := A+1] ∘ [ValF := ValF+1] ∘
  [WRITELN (A, ValF)] ∘ [VAR ValF: INTEGER]^T ∘ [WRITELN (A, B)] ∘
    [VAR A, B: INTEGER]^T ∘ [.] ({OUTPUT·<††,††,W>, A·0, B·0})
= [ValF := B] ∘ [A := A+1] ∘ [ValF := ValF+1] ∘ [WRITELN (A, ValF)] ∘
  [VAR ValF: INTEGER]^T ∘ [WRITELN (A, B)] ∘ [VAR A, B: INTEGER]^T ∘ [.]
    ({OUTPUT·<††,††,W>, A·0, B·0, ValF·?})
= [A := A+1] ∘ [ValF := ValF+1] ∘ [WRITELN (A, ValF)] ∘
  [VAR ValF: INTEGER]^T ∘ [WRITELN (A, B)] ∘
    [VAR A, B: INTEGER]^T ∘ [.] ({OUTPUT·<††,††,W>, A·0, B·0, ValF·0})
= [ValF := ValF+1] ∘ [WRITELN (A, ValF)] ∘ [VAR ValF: INTEGER]^T ∘
  [WRITELN (A, B)] ∘ [VAR A, B: INTEGER]^T ∘ [.]
    ({OUTPUT·<††,††,W>, A·1, B·0, ValF·?})
= [WRITELN (A, ValF)] ∘ [VAR ValF: INTEGER]^T ∘ [WRITELN (A, B)] ∘
  [VAR A, B: INTEGER]^T ∘ [.] ({OUTPUT·<††,††,W>, A·1, B·0, ValF·1})
= [VAR ValF: INTEGER]^T ∘ [WRITELN (A, B)] ∘ [VAR A, B: INTEGER]^T ∘ [.]
  ({OUTPUT·<†  1  1/†,††,W>, A·1, B·0, ValF·1})
= [WRITELN (A, B)] ∘ [VAR A, B: INTEGER]^T ∘ [.]
  ({OUTPUT·<†  1  1/†,††,W>, A·1, B·0})
= [VAR A, B: INTEGER]^T ∘ [.]
  ({OUTPUT·<†  1  1/  1  0/†,††,W>, A·1, B·0})
= [.] ({OUTPUT·<†  1  1/  1  0/†,††,W>})
= <†  1  1†, †  1  0†>
```

15.4.1 Exercises

15.4.1 Value parameters have been called "initialized local variables." Explain why.

15.4.2 Explain why the restriction that a constant identifier cannot be passed as a parameter applies only to variable parameters.

15.4.3 In the meaning given to procedure statements with parameters, explain why passing variable X as actual parameter to a procedure in a variable parameter requires renaming a local variable X in the procedure, but passing X in a value parameter does not.

15.5 Functions

Preview

Like procedures, functions encapsulate extensive computations so that they may be used easily, and may be modified by transmitting different arguments when used. However, functions return values, and are called by naming them within expressions.

Procedure declarations encapsulate sequences of operations, often to compute and return a value through a variable parameter. Function declarations and calls in Pascal provide another way of returning the computed value. Function declarations look much like procedure declarations, but the header ends with a colon and a type identifier, giving the type of the result. A procedure communicates its results by assigning values to its parameters. Within the block of a function declaration, the result of the function is indicated by assigning it to the identifier that is the name of the function. If no assignment is made to this identifier in the block, the result of the function invocation is undefined. The mechanism for returning results is illustrated by Max2 below:

```
FUNCTION Max2(P1, P2: INTEGER): INTEGER;
  {(P1 > P2 --> Max2 := P1) |
   (P2 >= P1 --> Max2 := P2)}
  BEGIN {Max2}
    IF P1 > P2
    THEN
      Max2 := P1 {return P1 as Max2's value}
    ELSE
      Max2 := P2 {return P2 as Max2's value}
  END;  {Max2}
```

The syntax rules for function definitions appear below:

<procedure and function declaration part> ::= <proc/func declarations> ¦

<proc/func declarations> ::= <proc/func declaration>
¦ <proc/func declarations> <proc/func declaration>

<proc/func declaration> ::= <heading> ; <block> ;

<heading> ::= PROCEDURE <identifier> <formal parameter list>
¦ FUNCTION <identifier> <formal parameter list> : <result type>

Function calls look like procedure calls: the name of the function is followed by a list of actual parameters, which are expressions, separated by commas. However, there is an important difference between function and procedure calls. Since functions deliver values, function calls are expressions which may appear within larger expressions or alone wherever an expression could appear in a program. Procedure calls, which do not deliver values, are statements. To find the maximum of three values, Max3 could be declared as follows, using two calls to Max2 as right sides of assignment statements.

```
FUNCTION Max3(P1, P2, P3: INTEGER): INTEGER;
  {Max3 is assigned the largest value from among
   P1, P2, and P3}
  VAR
    T: INTEGER;
  BEGIN {Max3}
    T := Max2(P2,P3);
    Max3 := Max2(P1,T)
  END;  {Max3}
```

When Max2 is invoked the first time it returns the larger of the values of P2 and P3, which is then assigned to T. T is subsequently used as an actual parameter in the the second invocation of Max2. Since function invocations can appear anywhere expressions can and Max2 declares its formal parameters to be value parameters, Max3 could be rewritten by making the first call to Max2 a parameter of the second call to Max2.

```
FUNCTION Max3(P1, P2, P3: INTEGER): INTEGER;
  {Max3 is assigned the largest value from among
   P1, P2, and P3}
  BEGIN {Max3}
    Max3 := Max2(P1, Max2(P2, P3))
  END;  {Max3}
```

The expression in the assignment statement would be evaluated by first calling Max2 to obtain the maximum of P2 and P3, and then calling Max2 again with P1 and the result of the earlier call.

The following rules of syntax incorporate function calls into expressions by adding them to <factor>.

<factor> ::= <variable access> ¦ <function designator>
¦ <unsigned constant> ¦ (<expression>) ¦ NOT <factor>

<function designator> ::= <function identifier> <actual parameter list>

<function identifier> ::= <identifier>

<actual parameter list> ::= (<actual parameter list>) ¦

<actual parameter list> ::= <actual parameter list> , <actual parameter>
¦ <actual parameter>

<actual parameter> ::= <expression>

Notice that the <actual parameter list> may be empty—a function may have zero parameters.

A function declaration adds the text of the function's <block> to the state under the identifier that names the function, just as for a procedure declaration. When the function is later called, this text is invoked to obtain a value. If there are variable parameters, or conflicts between local declarations and global identifiers in the state, the text must be modified as described in Sections 9.1.4 and 9.2.5, respectively. If there are value parameters, these must be treated as described in Section 15.4. Suppose a function F is declared as

```
FUNCTION F ... : Typ;
  T ;
```

with body T. Ignoring all modifications required for parameters and local variables, the meaning of a call on F in state s is:

$$\boxed{F}(s) = \boxed{F'}(\boxed{s(T')}(\boxed{\text{VAR } F'\text{: Typ}}(s))) =$$
$$(\boxed{\text{VAR } F'\text{: Typ}} \circ \boxed{s(T')} \circ \boxed{F'})(s).$$

Where F' is a new identifier, and T' is the same as T except that every occurrence of F on the left side of an assignment in T is replaced by F' in T'. That is, a temporary state is created by adding a new identifier F' to s; the modified body of the function acts on the temporary state; then the

value attached to F' is extracted from it. The value returned by a function is the most recent assignment to its identifier in the body; the declaration of a special identifier to receive this value is what makes the definition work.

This definition does not properly account for any modifications the function body may make in a state, because it happens to use identifiers that are VAR parameters or global variables. Such modifications are called *side effects*, and are treated in the next section.

As an example of a side-effect-free function call, consider a call on Max2 above:

```
WRITE('Larger is ', Max2(3, 7))
```

The parameters may be INTEGER constants, since Max2 has value parameters, and since a call on Max2 is an expression, it can appear in a WRITE statement. Suppose that the state at the time of this call is

$s = \{$OUTPUT·<††,††,**W**>, Max2·BEGIN IF P1 > P2 ... END$\}$.

Then

$$\boxed{\texttt{WRITE('Larger is ', Max2(3, 7))}}(s)$$
$$= \{\texttt{OUTPUT·<}\boxed{\texttt{Max2(3, 7)}}(s)\texttt{,††,W>, Max2·BEGIN ... END}\}.$$

From the definition above, with the modification required for value parameters,

$$\boxed{\texttt{Max2(3, 7)}}(s) = (\boxed{\texttt{VAR P1,P2: INTEGER; P1:=3; P2:=7}} \circ \boxed{\texttt{VAR Max2N: INTEGER; } B} \circ \boxed{\texttt{Max2N}})(s),$$

where Max2N is the new identifier, and B is s(Max2) modified to:

```
BEGIN
  IF P1 > P2
  THEN
    Max2N := P1
  ELSE
    Max2N := P2
END
```

by the insertion of Max2N. This is:

$$\boxed{\texttt{Max2(3, 7)}}(s) = \boxed{\texttt{Max2N}}(\boxed{\texttt{VAR P1,P2: INTEGER; P1:=3; P2:=7}} \circ \boxed{\texttt{VAR Max2N: INTEGER; } B})(s)$$
$$= \boxed{\texttt{Max2N}}(\boxed{B}(\{\texttt{P1·3, P2·7, Max2N·?,}$$
$$\texttt{OUTPUT·<††,††,W>, Max2·BEGIN ... END}\}$$
$$= \boxed{\texttt{Max2N}}(\{\texttt{P1·3, P2·7, Max2N·7, OUTPUT·<††,††,W>, Max2·BEGIN ... END}\}$$
$$= 7.$$

So the result is

$$\boxed{\texttt{WRITE('Larger is ', Max2(3, 7))}}(s)$$
$$= \{\texttt{OUTPUT·<†Larger is 7†,††,W>, Max2·BEGIN ... END}\}.$$

15.5.2 Side Effects in Functions

When a statement in a function body alters the value of a nonlocal variable or formal variable parameter, a call on the function (or the function itself) is said to have a *side effect*. The name arises because a function's *main* effect is to return a value. Side effects often come as a nasty surprise to the programmer, who may not be expecting any action from a function call other than the value returned. Furthermore, side effects in functions can make the results of expression evaluation unpredictable. The order in which operators are applied is defined by the Pascal syntax rules, but the language definition does not specify the order in which operands are fetched. Consider

the following definition, whose intention is given by the comment:

```
FUNCTION Identity(VAR In: INTEGER): INTEGER;
  {<Identity,In,Glob> := <In,In+1,Glob+3>}
  BEGIN {Identity}
    Identity := In;
    In := In + 1;
    Glob := Glob + 3
  END;  {Identity}
```

Its main effect is to return the value of the parameter `In`; it has side effects of altering this parameter and also altering the global variable `Glob`. The following values then can result from the expression

```
Glob + Identity(In) + In
```

depending on the way the operands are fetched:

Possible Values of `Glob + Identity(In) + In`

Value	Order of Evaluation

`Glob+In+In+1`	`{Glob and In fetched left to right:`
	`T1 := Glob;`
	`T2 := Identity(In);`
	`T3 := In;`
	`Result := T1 + T2 + T3 }`

`Glob+3+In+In`	`{Glob and In fetched right to left:`
	`T1 := In;`
	`T2 := Identity(In);`
	`T3 := Glob;`
	`Result := T3 + T2 + T1 }`

`Glob+In+In`	`{Glob and In fetched before the call:`
	`T1 := Glob;`
	`T2 := In;`
	`T3 := Identity(In);`
	`Result := T1 + T3 + T2 }`

`Glob+3+In+In+1`	`{Glob and In fetched after the call:`
	`T1 := Identity(In);`
	`T2 := Glob;`
	`T3 := In;`
	`Result := T2 + T1 + T3 }`

There are many cases in which the fetch order does not alter the value of an expression, but because of the potential surprise that any side effect may cause, it is good programming practice to avoid side effects in functions. When it is necessary to modify a variable parameter, a procedure is a better choice, and modification of global variables within functions or procedures should be avoided.

15.5.2 Predefined Functions

Several functions are *predefined* in the Pascal machine; these functions can be used without declaring them.

SUCC and PRED

Two predefined functions are available for all the ordinal types: successor (SUCC) and predecessor (PRED). SUCC and PRED do not have a standard definition for character expressions because of the differences in character sets on different machines. For the characters which correspond to digits, these functions work much as expected:

CExp	SUCC (CExp)	PRED (CExp)
0	1	undefined
1	2	0
2	3	1
	etc.	
9	undefined	8

However, SUCC('A') may not have value **B** on some Pascal machines, because the values **A** and **B** are not represented by adjacent codes in the underlying machine. Thus these built-in functions cannot sensibly be used with alphabetic characters on all machines, and the careful programmer avoids using them even if they work properly on today's machine.

Since only a subset of the integers can be represented in a Pascal program, the functions SUCC and PRED are defined for INTEGER expressions in the obvious way:

$$\boxed{\texttt{SUCC(}\ E\ \texttt{)}}(s) = \boxed{E\ \texttt{+}\ \texttt{1}}(s),$$

$$\boxed{\texttt{PRED(}\ E\ \texttt{)}}(s) = \boxed{E\ \texttt{-}\ \texttt{1}}(s),$$

where these equations include the possibility that the value of E is $\pm MAXINT$, and the corresponding addition or subtraction is undefined.

SUCC and PRED can be applied to enumerated-type operands to produce values in the sequence defined by the declaration. For example, consider the declaration:

```
TYPE
  Month = (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);
```

The order of appearance of the identifiers in the type declaration is from the constant with the least value to that with the greatest value. For example:

Expression	Value
SUCC(Jan)	Feb
PRED(Oct)	Sept
PRED(NoMonth)	undefined
SUCC(Dec)	undefined

Enumerated types can be used to eliminate "mystery" numbers from programs. Mystery numbers are those that arise from arbitrary mappings of real-world values to numerical values. If the programmer had made up integer values for the months instead of declaring TYPE Month, 0 for January, 1 for February, etc., a program to list the number of days in each month might be:

```
PROGRAM PrintDays(INPUT, OUTPUT);
  VAR
    Mo: INTEGER;
  BEGIN {PrintDays}
    Mo := 0;
    WHILE Mo <= 11
    DO
      BEGIN
        IF (Mo=3) OR (Mo=5) OR (Mo=8) OR (Mo=10)
        THEN
          WRITELN(30)
        ELSE
          IF (Mo=0) OR (Mo=2) OR (Mo=4) OR (Mo=6)
             OR (Mo=7) OR (Mo=9) OR (Mo=11)
          THEN
            WRITELN(31)
          ELSE
            WRITELN(28);
      Mo := Mo + 1
      END
  END.  {PrintDays}
```

Execution

```
OUTPUT:        31
               28
               31
               30
               31
               30
               31
               31
               30
               31
               30
               31
```

Mystery numbers make programs hard to read, and when (say) 1 appears it may represent February, or perhaps a page number of some report, or the first day of some month, etc.

A program using an enumerated type for the same purpose is better:

```
PROGRAM PrintDays(INPUT, OUTPUT);
  VAR
    Mo: (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
         Jul, Aug, Sep, Oct, Nov, Dec);
  BEGIN {PrintDays}
    WRITELN(31);
    Mo := Jan;
    WHILE Mo < Dec
    DO
      BEGIN
        Mo := SUCC(Mo);
        IF (Mo=Sep) OR (Mo=Apr) OR (Mo=Jun) OR (Mo=Nov)
        THEN
          WRITELN(30)
        ELSE
          IF Mo=Feb
```

```
        THEN
          WRITELN(28)
        ELSE
          WRITELN(31)
    END
END.  {PrintDays}
```

The program using an enumerated type is easier to read because the constant identifiers are more meaningful than the integers in the previous version of the program. Instead of having to count down to discover that September is 8, Sep comes immediately to mind. It is easier to use the mnemonic rhyme "Thirty days hath September, April, ..." in the program with an enumerated type.

The WHILE statement also exhibits the use of a relational operator and SUCC with values from an enumerated type. The WHILE statement body must not be executed when Mo has the value Dec because SUCC is undefined on the final value of an enumerated type. Thus, the following shorter incorrect program fragment must be avoided in favor of the longer, correct version.

```
{Incorrect Version}

Mo := Jan;
WHILE Mo <= Dec
DO
  BEGIN
    {statement list};
    Mo := SUCC(Mo)
  END
```

```
{Correct Version}

Mo := Jan;
{statement list};
WHILE Mo < Dec
DO
  BEGIN
    Mo := SUCC(Mo);
    {statement list}
  END
```

The WHILE statement was "unrolled" by repeating its body to handle the case that Mo has the value Jan, and incrementing Mo before executing the body in the BEGIN statement. Thus, on the last execution of the BEGIN statement, Mo has the value Dec and the next test of the WHILE condition fails. (Another way to solve this problem would be to move NoMonth to the end of the enumeration, providing a phony month after Dec.)

SUCC and PRED are not often used with the BOOLEAN type, but the results can be predicted from the information that this type acts as if it had been declared:

```
TYPE
  BOOLEAN = (FALSE, TRUE)
```

ODD, ABS, and SQR

These three functions take INTEGER arguments. ODD returns value TRUE if the argument is odd (i.e., a member of {..., -3, -1, 1, 3, 5, ...}), FALSE if the argument is even. ABS returns an INTEGER value that is the absolute value of the argument, and SQR produces the INTEGER square of the argument value. For example:

Expression	Value
ODD(-5)	TRUE
ODD(10)	FALSE
ABS(-5)	5
ABS(12)	12
SQR(-5)	25
SQR(10)	100

ORD and CHR

ORD and CHR are built-in functions that give access to the numerical position of values in finite enumerations. They are most often used to manipulate characters as if they were integers. ORD converts the first value of an enumerated type to 0, the second to 1, etc. Thus for example in the Month type

```
(NoMonth, Jan, Feb, Mar, Apr, May, Jun,
 Jul, Aug, Sep, Oct, Nov, Dec)
```

ORD(Jun) has value 6.

When ORD is applied to a CHAR argument it returns the position of the character in the collating sequence of the Pascal machine. CHR is the inverse: on argument a collating-sequence number, it returns the corresponding character. For example, CHR(ORD('F')) has value **F**. It is not particularly useful to work with the numerical value of a character unless its position relative to other characters is known. Since the digit characters are guaranteed to fall in sequence,

```
ORD('7') - ORD('0')
```

has INTEGER value 7—that is, this expression can be used to convert a digit character to its corresponding integer value. (Subtracting the position of **0** in the sequence would not be necessary if the character **0** occupied position zero, but it does not on most Pascal machines.) To go the other way requires both ORD and CHR, for example to convert the integer value 7 to **7**:

```
CHR(7 + ORD('0'))
```

Programmers who are more comfortable with the INTEGER type than with enumerated types often use ORD to excess, and thereby lose the advantages of the mnemonic constants.

15.0.1 Exercises

15.0.0 A function call and a procedure call have the same syntactic form—an identifier followed by an optional list of actual parameters. Can the two forms be used interchangeably? How is your answer reflected in the syntax rules?

15.0.0 Suppose that Max2 (Section 15.5) had been written with variable parameters instead of value parameters. Explain why such a Max2 could not be used as an argument to itself in the second version of Max3.

15.0.0 Consider a Pascal FUNCTION that does digit input. Its header is

```
FUNCTION DigitR: INTEGER;
```

and each time it is called, the next character is read from INPUT, converted from CHAR to INTEGER, and returned. If the next character is not a digit, -1 is returned. For example, if INPUT contains the string †**123X**† then the value of

```
DigitR + DigitR + DigitR + DigitR
```

is 5.

a) Write DigitR using ORD.

b) Write DigitR without using ORD.

c) Use DigitR to write another function that reads strings for base-ten numbers in a fixed range, returning the number if in range, or -1 if out of range:

```
FUNCTION Base10(MaxVal: INTEGER): INTEGER;
{Interpret a string of digits followed by a nondigit
 as a representation of a base-ten integer N.
 If 0 <= N <= MaxVal, return the value; otherwise,
 return -1.}
```

d) Are there side effects in DigitR? In Base10? How could side effects cause a programmer trouble in this case?

15.5.4 Each Pascal fragment below contains the identifier Qu. You are to deduce a possible type of Qu from the context in each separate case. Express the answer as a type specification for the declaration:

```
VAR
  Qu: ?
```

Replace the question mark with the type required of Qu. For example, if you decide that one answer is a subrange of CHAR between **3** and **6**, fill in:

```
VAR
  Qu: '3'..'6'
```

If there are several possible types, explain briefly.

a) `Qu MOD PRED(Qu)`

b) `SUCC(Qu > '8')`

c) `CHR(ORD(SUCC(Qu)))`

15.5.5 What values are printed by the following programs?

a)
```
PROGRAM Alias1(OUTPUT);
  VAR
    X: INTEGER;
  PROCEDURE P(VAR A,B: INTEGER; C: INTEGER);
    BEGIN {P}
      A := A + 1;
      B := B * A;
      C := C + X;
      WRITELN(A,B,C)
    END;  {P}
  BEGIN {Alias1}
    X := 1;
    P(X,X,X);
    WRITELN(X)
  END.  {Alias1}
```

b)
```
PROGRAM Alias2(OUTPUT);
  VAR
    X, Y: INTEGER;
  PROCEDURE P(VAR A,B: INTEGER; C: INTEGER);
    BEGIN {P}
      A := A + 1;
      B := B * C;
      C := X * Y;
      WRITELN(A,B,C)
    END;  {P}
  BEGIN {Alias2}
    X := 1; Y := 2;
    P(X,Y,X);
    WRITELN(X,Y)
  END.  {Alias2}
```

15.5.6 Write a program to print out the absolute sequence positions of the characters **0**...**9a**...**zA**...**Z** in your Pascal machine. Explain why it is or is not all right to use SUCC to pass from character to character in the program.

15.5.7 Explain why the formal definition of function-call meaning in Section 15.5 is not the proper intuitive meaning when there are side effects. Give an example of a call where the meaning is wrong, showing both what the definition provides, and what it should provide.

15.5.8 Formulate a formal definition for the meaning of a function call with side effects. (Hint: make the function call into an equivalent procedure call.)

15.5.9 Consider the following declaration:

```
FUNCTION Func(VAR Formal: INTEGER);
  BEGIN {Func}
    Formal := Formal + 1;
    Func := Formal
  END;  {Func}
```

a) Rewrite it as a procedure, retaining the nonfunctional behavior of modifying the parameter.

b) Rewrite the function, to obtain the effect of computing the mathematical "add one" function without side effects. Can this be done without changing the variable parameter to a value parameter?

15.5.10 The detrimental aspect of side effects of function and procedure calls is that the effect is hidden: it literally happens "on the side" and the programmer may not expect it. Listed below are several locations in Pascal programs. Mark each one SAFE or DANGER as to whether a side effect could occur there and cause trouble for an unsuspecting programmer. The first one is done for you: a procedure statement is obviously a dangerous position because procedures have only side effects. If you are in doubt about an answer, explain briefly.

DANGER In a procedure statement
_____ In the test part of an IF statement
_____ In the right side of an assignment statement
_____ In a WRITE statement
_____ In an actual parameter being passed by value

15.5.11 Consider a function that is part of a data abstraction module, intended to be called from outside the module. It is possible for the function to have a side effect, yet not violate the rules of separation between module and users. Explain how this can be.

15.5.12 Given the specification

$R = \{(0,0), (0,1), (1,0), (1,1), (2,4), (2,1)\}$

determine whether each of the functions of X below is correct with respect to R.

a) (X>=0 → X := X*X)

b) (X ∈ {1, 2, ..., 255} → X := ORD(CHR(X))-1)

(where CHR is defined in the range [1, 255])

c) (X ∈ INTEGER → X := SUCC(X))

15.5.13 Ignoring the effects on K and Z, show the consistency of the specification:

(N>=0 AND X<>0 → Y := X^N)

and the program fragment:

```
K := N; Y := 1; Z := X;
WHILE K <> 0
DO
  IF ODD(K)
  THEN
    BEGIN
      K := K - 1; Y := Y * Z
    END
  ELSE
    BEGIN
      K := K DIV 2; Z := Z * Z
    END
```

15.6 Forward Declarations

Preview

Pascal's rule that identifiers must be declared before use is saved in the case of mutual recursion by the FORWARD declaration.

With mutually recursive procedures (e.g., two procedures each calling the other) Pascal's rule to declare identifiers before using them cannot be observed. For example, the call of Second precedes the declaration of Second in the following example, but moving Second before First will only result in the call of First preceding its declaration.

```
PROCEDURE First(IVar: INTEGER);
  BEGIN {First}
    ...;
    Second(CHR(IVar));
    ...
  END;  {First}
PROCEDURE Second(CVar: CHAR);
  BEGIN {Second}
    ...;
    First(ORD(CVar));
    ...
  END   {Second}
```

Pascal provides a way to preserve its rule of declaration before use in the FORWARD declaration. One of the mutually recursive procedures has its heading moved up, but its body remains in place. A FORWARD declaration includes the entire procedure heading (name, formal parameter list, return type), but replaces the block with the reserved word FORWARD. The procedure's block appears later in the program, but only the reserved word PROCEDURE or FUNCTION and the name of the procedure may appear with the block. The parameters and return type are not repeated. For the example above:

```
PROCEDURE Second(CVar: CHAR); FORWARD;
PROCEDURE First(IVar: INTEGER);
  BEGIN {First}
    ...;
    Second(CHR(IVar));
    ...
  END;  {First}
PROCEDURE Second; {don't repeat parameters}
  BEGIN {Second}
    ...;
    First(ORD(CVar));
    ...
  END   {Second}
```

15.7 Summary

Several declarations were introduced in this chapter: external files, constants, types, value parameters, and functions. External files provide a way to create program systems. Constant and type declarations help organize programs by centralizing information, making a program easier to change. For example, consider the following:

```
TYPE
  T = ...; {any type with > and := operations}
FUNCTION Max2(VAR P1, P2: T): T;
  {(P1 > P2 --> Max2 := P1) |
   (P2 >= P1 --> Max2 := P2)}
  BEGIN {Max2}
    IF P1 > P2
    THEN
      Max2 := P1
    ELSE
      Max2 := P2
  END;  {Max2}
```

By simply changing the definition of T (say from CHAR to INTEGER), Max2 could be applied to integers rather than characters.

Pascal procedures may have value parameters. Value parameters are more secure than variable parameters because inadvertent assignments to them do not change the execution state where the call was made. Thus, input values are passed as value parameters, and output variables or variables to be updated are passed as variable parameters.

Like procedures, functions help structure programs by abstracting behavior. However, function calls are expressions that return values, while procedure calls are statements that act by changing either their variable formal parameters or some nonlocal variables. Also, since functions can only return a single value, procedures must be used to return multiple values through assignments to variable parameters.

CHAPTER 16

AGGREGATE DATA TYPES IN D PASCAL

Chapter Preview

This chapter describes three mechanisms for creating composite values in Pascal: sets, files and records. A limited version of set objects is provided. Files extend the TEXT files of CF Pascal to types other than CHAR. Records providing a grouping mechanism for any finite collection of types.

Each ordinal type contains values that are atomic—the values are not composite. Aggregate types, on the other hand, combine previously defined types, which may be ordinal or other aggregate types, into new types that contain components of the previously defined types.

<new type> ::= <new ordinal type> ¦ <aggregate type>

<aggregate type> ::= <unpacked structured type>

<unpacked structured type> ::= SET OF <base type>
¦ FILE OF <component type>
¦ RECORD <field list> END

<base type> ::= <ordinal type>

<ordinal type> ::= <new ordinal type> ¦ <ordinal type identifier>

<ordinal type identifier> ::= <type identifier>

<component type> ::= <type denoter>

Sets, files, and records combine component types in different ways. Set and file types group identical components, while record types group components that differ. The set and record types group components without regard to order, while file types form sequences of components. Any component of a record may be selected; the components of a file may only be accessed sequentially; and no selection operation at all is defined for set components.

16.1 Sets

Preview

Pascal's set data type allows the programmer to declare and use variables that are sets whose members are drawn from (small) finite collections. A full range of set operations is available.

A set type is declared as follows:

```
TYPE
  SetType = SET OF ComponentType;
```

where ComponentType is an ordinal type. Pascal sets are further restricted to take their members from a limited collection, whose maximum size is not part of the definition of Pascal but varies from one machine to another. (The limit is typically small, often only a few hundred.) Thus, it is illegal to declare:

```
TYPE
  IntSet = SET OF INTEGER;
```

but legal to declare:

```
CONST
  Max = 20;
TYPE
  IntSet = SET OF 2..Max;
```

if `Max` is chosen small enough to be within the limit. Declaration statements do not provide an initial value for a set variable. Thus, the value of `Sieve` after

```
VAR
  Sieve: SET OF 2..Max;
```

is undefined.

Set constants are constructed by enclosing lists or subranges of expressions in square brackets. A set constant containing an empty list of values (i.e., `[ ]`) is used to represent the empty set.

The possible values for a set are the members of the powerset (i.e., set of all subsets) of the collection of values used to define the type. For example, the powerset of the type:

```
TYPE
  SmallIntSet = SET OF 1..3
```

is

{{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}},

so any of these sets may be values of type `SmallIntSet`. The Pascal set constant for the value set that has three elements would be written `[1,2,3]` or `[1..3]`.

Any of the following assignment statements initialize `Sieve` to the set value {2,3,...19,20}:

```
Sieve := [2..20];
Sieve := [2,3,4,5,6,7,8,9,10,11,12,13,
          14,15,16,17,18,19,20];
Sieve := [2..9,10,11..15,16,17,18..20]
```

The operations on set values are `+` (set union), `-` (set difference), `*` (set intersection); `=` (equality), `<>` (inequality), `<=` (nonstrict subset), `>=` (nonstrict superset); and `IN` (set membership).

Set Operator	Operands → Result
`+  *  -`	Set × Set → Set
`=  <>  <=  >=`	Set × Set → Boolean
`IN`	ComponentType × Set → Boolean
`[ ]`	ComponentType × ... × ComponentType → Set

The union, difference, and intersection operations are defined only for pairs of sets, not set components. Thus if `Sieve` has integer members it is illegal to write:

```
Sieve + 2
```

Instead, the element must be made into a 1-set value:

```
Sieve + [2]
```

Some sample relational expressions and their values are:

Expression	Value
`[3, 5] <= [3 .. 5]`	TRUE
`[3 .. 5] <= [3 .. 6]`	TRUE
`[3 .. 5] >= [3 .. 5]`	TRUE
`[3 .. 5] >= [3 .. 6]`	FALSE

The binary infix operator `IN` tests set membership:

Expression	Value
`4 IN [3 .. 5]`	TRUE
`4 IN []`	FALSE
`4 IN [3, 5]`	FALSE

The square brackets are not only used for set constants, but to form sets from expressions whose values are the member elements. For example, if `X` is of type `INTEGER` with value 3:

Expression	Value
`[2, X]`	{2, 3}
`[3..X]`	{3}
`[X .. X+2, SUCC(X+7)]`	{3, 4, 5, 11}
`[X..1]`	{ }

Properties of the set operators are shown below:

Operator	Associativity	Precedence
`[ ]`	none	highest
`*`	left	↑
`+ -`	left	⋮
relationals	none	lowest

Evaluations of some sample set values are shown below in the Pascal constants notation.

```
[1] + [7 .. 9] * [7, 9]  =  [1] + [7, 9]  =  [1, 7, 9]
[4 .. 6] - [4, 6] + [1]  =  [5] + [1]     =  [5, 1]
```

16.1.1 Counting Characters

Sets can simplify some of the tests in programs. For example, the following program counts the number of letter and digit characters in the input.

```
PROGRAM AlphaCount(INPUT, OUTPUT);
  VAR
    Ch: CHAR;
    Count: INTEGER;
  BEGIN {AlphaCount}
    Count := 0;
    WHILE NOT EOF
    DO
      BEGIN
        READ(Ch);
        IF ((Ch >= 'a') AND (Ch <= 'z')) OR
           ((Ch >= 'A') AND (Ch <= 'Z')) OR
           ((Ch >= '0') AND (Ch <= '9'))
        THEN
          Count := Count + 1
      END;
    WRITELN(Count, ' alphanumeric characters read')
  END.  {AlphaCount}
```

Executions

```
INPUT :abc U de01 23f
OUTPUT:          11 alphanumeric characters read
INPUT :
OUTPUT:           0 alphanumeric characters read
```

The test for alphabetic characters needs no logical operations if a set is used, as in the fragment:

```
WHILE NOT EOF
DO
  BEGIN
    READ(Ch);
    IF Ch IN ['A'..'Z','a'..'z','0'..'9']
    THEN
      Count := Count + 1
  END;
```

16.1.2 Rules of Syntax and Meaning

The syntax rules for expressions including set operations are shown below.

<expression> ::= <simple expression>
  ¦ <simple expression> <relational operator> <simple expression>

<simple expression> ::= <simple expression> <add operator> <term>
  ¦ <term> ¦ <sign> <term>

<term> ::= <term> <mult operator> <factor> ¦ <factor>

<factor> ::= <identifier> ¦ <function designator>
  ¦ <unsigned constant> ¦ (<expression>)
  ¦ NOT <factor> ¦ <set constructor>

<relational operator> ::= = ¦ <> ¦ < ¦ <= ¦ > ¦ >= ¦ IN

<sign> ::= + ¦ -

<add operator> ::= + ¦ - ¦ OR

<mult operator> ::= * ¦ DIV ¦ MOD ¦ AND

<set constructor> ::= [<member designators>]

<member designators> ::= <member designators> , <member designator> ¦

<member designator> ::= <expression> ¦ <expression> .. <expression>

The meaning of each operation between set operands E1 and E2 and set-member operand Me can be given in terms of the usual mathematical operators as follows:

$[E1 = E2](s) = ([E1](s) = [E2](s))$

$[E1 <> E2](s) = ([E1](s) \neq [E2](s))$

$[E1 <= E2](s) = ([E1](s) \subseteq [E2](s))$

$[E1 >= E2](s) = ([E1](s) \supseteq [E2](s))$

$[Me\ IN\ E2](s) = ([Me](s) \in [E2](s))$

$[E1 + E2](s) = ([E1](s) \cup [E2](s))$

$[E1 - E2](s) = ([E1](s) - [E2](s))$

$\boxed{\texttt{E1 * E2}}(s) = (\boxed{\texttt{E1}}(s) \cap \boxed{\texttt{E2}}(s))$

$\boxed{\texttt{[Me]}}(s) = \{\boxed{\texttt{Me}}(s)\}$

$\boxed{\texttt{(E)}}(s) = \boxed{\texttt{E}}(s)$

for any state s. For more complex expressions the rules of syntax, associativity, and precedence specify the order in which operations are to be performed.

16.1.3 Prime Numbers

A method known as the sieve of Eratosthenes can be used to print the prime numbers between 2 and an arbitrary upper bound, say 16. Initially all the numbers between 2 and 16 are placed in the sieve. The smallest number in the sieve is prime; and all its multiples are removed from the sieve. The new smallest number in the sieve is the next prime and its multiples are also removed. This process is repeated until the maximum value in the sieve is reached. Initially, let

$S = \{2,3,4,5,6,7,8,9,10,11,12,13,14,15,16\}$.

Removing multiples of 2:

$S - \{2, 4, 6, 8, 10, 12, 14, 16\} = \{3, 5, 7, 9, 11, 13, 15\}$.

Multiples of 3:

$\{3, 5, 7, 9, 11, 13, 15\} - \{3, 6, 9, 12, 15\} = \{5, 7, 11, 13\}$.

(Some multiples of 3 have already been removed, i.e., those that are also multiples of 2.)

The series of set differences performed would be:

Contents of sieve	removed	multiples of
{2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}		
{3,5,7,9,11,13,15}	{2,4,6,8,10,12,14,16}	2
{5,7,11,13}	{3,6,9,12,15}	3
{7,11,13}	{5,10,15}	5
{11,13}	{7,14}	7
{13}	{11}	11
{}	{13}	13

The sieve is now empty and the primes up to 16 are 2, 3, 5, 7, 11, 13.

The set type is exactly the right data type to implement this algorithm. In the design of a solution for the prime number problem, the sieve is represented as a set of integers in order to use the set difference operator to remove elements from the set.

Design part 1

```
PROGRAM Prime(OUTPUT);
{Print the prime numbers between 2 and Max}
  CONST
    Max = 16;
  VAR
    Sieve: SET OF 2 .. Max;
    Next, NextMult: INTEGER;
  BEGIN {Prime}
    Sieve := [2 .. Max];
    Next := 2;
    WHILE Sieve <> []
    DO
      BEGIN
        {print smallest member Next of Sieve};
        {remove multiples NextMult of Next from Sieve}
```

```
      END
    END.  {Prime}
```

To find the smallest member of Sieve, since sets are unordered collections of values, requires testing successive integers.

Design part 1.1

```
BEGIN {print smallest member Next of Sieve};
  WHILE NOT(Next IN Sieve)
  DO
    Next := Next + 1;
  WRITE(Next:5)
END
```

Design part 1.2

```
BEGIN {remove multiples of Next from Sieve}
  NextMult := Next;
  WHILE NextMult <= Max
  DO
    BEGIN
      Sieve := Sieve - [NextMult];
      NextMult := NextMult + Next
    END
END
```

Assembling the program:

```
PROGRAM Prime(OUTPUT);
{Print the prime numbers between 2 and Max}
  CONST
    Max = 16;
  VAR
    Sieve: SET OF 2 .. Max;
    Next, NextMult: INTEGER;
  BEGIN {Prime}
    Sieve := [2 .. Max];
    Next := 2;
    WHILE Sieve <> []
    DO
      BEGIN
        BEGIN {print smallest member Next of Sieve}
          WHILE NOT(Next IN Sieve)
          DO
            Next := Next + 1;
          WRITE(Next:5)
        END;
        BEGIN {remove multiples of Next from Sieve}
          NextMult := Next;
          WHILE NextMult <= Max
          DO
            BEGIN
              Sieve := Sieve - [NextMult];
              NextMult := NextMult + Next
            END
        END
      END
  END.  {Prime}
```

Execution

```
OUTPUT:    2     3     5     7    11    13
```

Notice the use of the constant identifier `Max` in `Prime`. To change the upper bound on the range of numbers considered, only the constant definition need be changed.

16.1.4 Exercises

16.1.1 Explain why sets would not be very useful on a Pascal machine that limited the underlying type to have less than 26 elements.

16.1.2 How many set values of the type `SET of BOOLEAN` are there? How many members has the set of this type with the most members?

16.1.3 Suppose that for the declaration

```
VAR
  SetSized: SET of Typ
```

the underlying type `Typ` has exactly five possible values.

a) How many possible different values are there for `SetSized`?

b) Explain why no value of `SetSized` can have more than five members.

16.1.4 Rewrite the program `PrintDays` of Section 15.5.2 so that it tests for the 30-day months using a constant of type `SET of Month`.

16.1.5 In the context of the following program fragment:

```
VAR
  A: SET OF 1..20;
  B,C,D: INTEGER;
BEGIN
  A := [1..20];
  B := 5;
  C := 8;
  D := 10;
```

evaluate the following expressions:

a) `NOT([B,C,D] <= A)`

b) `[B..C] + [D] * [2..8]`

c) `[B..C] - A`

d) `B IN A`

e) `B IN (A - [B])`

f) `[B..C] * [C..D]`

16.1.6 Each Pascal fragment below contains the identifier `Qu`. You are to deduce a possible type of `Qu` from the context in each separate case. Express the answer as a type specification for the declaration:

```
VAR
  Qu: ?
```

Replace the question mark with the type required of `Qu`. For example, if you decide that one answer is a subrange of `CHAR` between **3** and **6**, fill in:

```
VAR
  Qu: '3'..'6'
```

If there are several possible types, explain briefly.

a) `'A' IN Qu`

b) `Qu + [3]`

16.1.7 Show that the following program fragment:

```
S := []; I := 0; N := 4;
WHILE I <= N
DO
  BEGIN
    I := I + 1;
    S := S + [I]
  END
```

computes the function:

```
S,I,N := [1..5],5,4
```

You may assume that the function of the WHILE statement is:

```
(I<=N → S,I := {I+k | 1≤k≤N-I+1}, N+1) |
(I>N →  )
```

16.1.8 Determine the function of the following WHILE statement.

```
WHILE B < 50
DO
  BEGIN
    A := A + [2*B];
    B := B + 1
  END
```

16.2 Files

Preview

Files are ordered sequences of values. WRITE statements append a value to an existing sequence (always at the end), and READ statements can obtain the values in the same order they were earlier written. The values are not limited to CHAR as in TEXT files, but may be of almost any Pascal type. Buffer variables and PUT/GET statements provide an additional way to manipulate files.

In addition to character files, Pascal lets us define new file types with components of any previously defined type other than file or another aggregate type containing a file as a component. A typical type declaration for a file is:

```
VAR
  IntFile: FILE OF INTEGER;
```

An `IntFile` value is a sequence of integer values—only `INTEGER` quantities may be read from, or written to, the file. File variables are the only Pascal variables that cannot be the target of assignment statements. All file values must be created using predefined operations.

A file declaration implicitly declares a *buffer variable* written as the file identifier followed by an up-arrow (e.g., `IntFile↑`) with the same type as the components of the file. When we are reading a file, `IntFile↑` contains the value currently available for reading. New file components are appended to a file by filling the buffer variable and writing its contents. Buffer variables can appear anywhere other variables can appear, e.g., in expressions:

```
3 + IntFile↑ * 7
```

However, buffer variables are not identifiers since they end in a character which is neither a letter nor a digit. Thus, we must update our syntax rules, replacing <identifier> on the right side of the

<factor> rule with <variable access> which derives both identifiers and buffer variables.

<factor> ::= <variable access> ¦ <function designator>
¦ <unsigned constant> ¦ (<expression>)
¦ NOT <factor> ¦ <set constructor>

<variable access> ::= <variable identifier> ¦ <buffer variable>

<buffer variable> ::= <file variable> ↑

<file variable> ::= <variable access>

(Although the syntax rules indicate that a file identifier followed by an arbitrary number of ↑ characters is a file variable, only a single ↑ may appear after the identifier.)

TEXT file values are 3-lists whose first two elements are strings for the past and future values of the file. For arbitrary files, the past and future values are lists of elements, elements of the type defined for the file. The third element of a file value is the mode indicator telling whether the file is open for reading (**R**) or writing (**W**).

Since the declaration of a file variable introduces a buffer variable in addition to the file, the semantics of the declaration statement for files must be altered to add a buffer variable with an undefined value to the state:

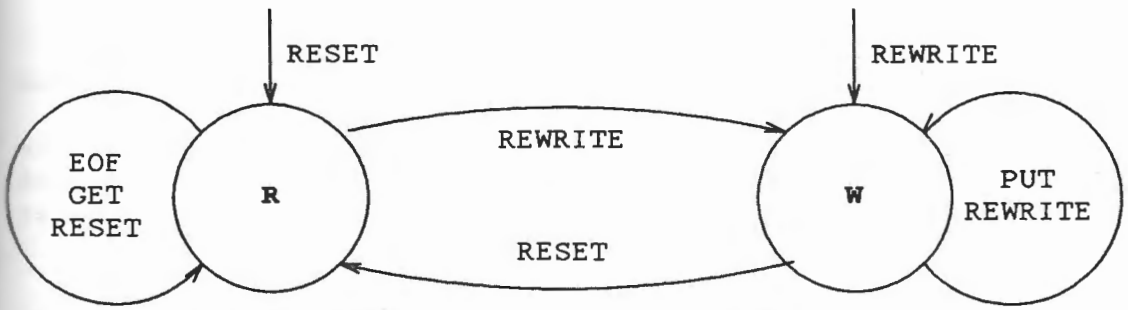
$\boxed{\texttt{VAR F: FILE OF TypeT}}(s) = s \cup \{F \cdot \langle ?,?,? \rangle,\ F\uparrow \cdot ?\}.$

16.2.1 Non-TEXT Files

Files are sequences of elements all of the same type. The operations on files are listed below.

Operation	Operand type → Result type
REWRITE	File → void
PUT	File → void
RESET	File → void
GET	File → void
EOF	File → Boolean

The term "void" is used to indicate that these operations do not return values; the operations are similar to Pascal procedure statements and can appear anywhere a statement can. The effects of each of these operations on the state of a file are indicated by the finite-state automaton below. Its states are the two file modes **R** and **W**.



An initial RESET or REWRITE puts a file in the **R** or **W** mode.

REWRITE and PUT are used to build file values. REWRITE empties a file and prepares it for writing.

$\boxed{\texttt{REWRITE(F)}} = \{(u, v)$: F is not INPUT or OUTPUT and $v = u$ except that $v(F) = \langle\langle\rangle,\langle\rangle,\mathbf{W}\rangle\}$.

If a file is open for writing, PUT appends the value of its buffer variable to the file and leaves the buffer variable undefined.

$\boxed{\texttt{PUT(F)}} = \{(u, v)$: $u(\texttt{F}).3 = \texttt{W}$ and $v = u$ except that
$v(\texttt{F}).1 = u(\texttt{F}).1\ \nabla(u(\texttt{F}\uparrow))$ and $v(\texttt{F}\uparrow)$ is undefined$\}$.

Thus PUT is undefined when a file is not open for writing.

For example, to write a file of integers, the file has to be declared and rewritten:

```
VAR
  IntList: FILE OF INTEGER;
BEGIN
  REWRITE(IntList);
  ...
END
```

Then values are assigned to IntList↑ so that subsequent PUT statements will append the values to IntList:

```
{equivalent of WRITE(IntList, 1)}
IntList↑ := 1;
PUT(IntList);   {appends 1 to IntList, IntList↑ undefined}
{equivalent of WRITE(IntList, 2)}
IntList↑ := 2;
PUT(IntList)    {appends 2 to IntList, IntList↑ undefined}
```

At this point, IntList contains the values 1 and 2 in its past list, the file is writable, and the contents of IntList↑ are not defined.

IntList = < <1,2>,<>,W > and IntList↑ = undefined

RESET and GET are the operations that retrieve values from files. RESET returns its file to its starting position so that reading can begin. The file's buffer variable is assigned the value of the first component of the file's future list (if one exists).

$\boxed{\texttt{RESET(F)}} = \{(u, v)$: F is not INPUT or OUTPUT and $v = u$ except that
$v(\texttt{F}) = <<>, u(\texttt{F}).1\ \&\ u(\texttt{F}).2, \texttt{R}>$ and $v(\texttt{F}\uparrow) = \Theta(u(\texttt{F}).1\ \&\ u(\texttt{F}).2)\}$.

Since the head function Θ is undefined for empty lists, the buffer variable is undefined when RESET is applied to empty files.

If a file is open for reading and its future list is not empty, GET transfers the first element from the future list to the past list, and assigns the value of the new first element of the future list to the file's buffer variable.

$\boxed{\texttt{GET(F)}} = \{(u,v)$: $u(\texttt{F}).3 = \texttt{R}$ and $u(\texttt{F}).2 \neq <>$ and $v = u$ except that
$v(\texttt{F}).1 = u(\texttt{F}).1\ \nabla(\Theta(u(\texttt{F}).2))$, $v(\texttt{F}).2 = \Lambda(u(\texttt{F}).2)$, and
$v(\texttt{F}\uparrow) = \Theta(\Lambda(u(\texttt{F}).2))\}$.

If a file is not open for reading, the statement aborts and program execution will halt. The statement will also abort if there is no data left to read (i.e., the future list is empty).

The file IntList constructed above has two integer values in its past list and is open for writing; that is, it's value is $<<1,2>, <>, \texttt{W}>$. A RESET operation changes the values of both IntList and IntList↑ to $<<>, <1,2>, \texttt{R}>$ and 1, respectively. Now a GET on IntList moves 1 to the past list of the file so the value of IntList becomes:

$$<\nabla(\Theta(<1,2>)),\ \Lambda(<1,2>),\ \texttt{R}> = <<1>, <2>, \texttt{R}>,$$

and the buffer pointer IntList↑ has value

$$\Theta(\Lambda(<1,2>)) = \Theta(<2>) = 2.$$

The end-of-file function EOF returns a Boolean value: TRUE if no more components remain in the file's future list and FALSE otherwise.

$$\boxed{\texttt{EOF(F)}}(u) = (u(\texttt{F}).2 = <>)\ .$$

If EOF is true, the value in the file buffer may not be accessed because it is undefined.

EOF(IntList) is FALSE after the first GET described above, since IntList has a future list containing 2. A second GET makes the file value

$$\langle\langle 1\rangle\ \nabla(\Theta(\langle 2\rangle)),\ \Lambda(\langle 2\rangle),\ \mathbf{R}\rangle = \langle\langle 1,2\rangle,\ \langle\rangle,\ \mathbf{R}\rangle$$

and the buffer-variable value

$$\Theta(\Lambda(\langle 2\rangle)) = \Theta(\langle\rangle)$$

which is undefined. The value of EOF(IntList) is now TRUE. To avoid an attempt to access a buffer variable when it is undefined we adopt programming paradigms in which EOF is always tested before the buffer is accessed. For example:

```
RESET(F);
WHILE NOT(EOF(F)) DO
  BEGIN
    ...  F↑ ... {access the buffer variable}
    .
    .
    GET(F)
  END
```

16.2.2 OddBeforeEven Program System

Pascal TEXT files can usually be created without using a PASCAL program, for example by typing characters using an editor. But a FILE OF INTEGER (say) cannot be created by typing what look like integer values using an editor. Such a file is still a TEXT file, in which characters like **3** appear. To create non-TEXT files Pascal must be used. For example, the following program reads the TEXT input file and creates a file with integer values.

```
PROGRAM GenIntFile(INPUT, IntIn, OUTPUT);
  {write the integers in INPUT to IntIn,
   echo INPUT to OUTPUT}
  VAR
    IntIn: FILE OF INTEGER;
    Int: INTEGER;
  BEGIN {GenIntFile}
    REWRITE(IntIn);
    READ(Int); {converts INPUT chars to integer}
    WHILE Int <> -1
    DO
      BEGIN
        {WRITE(IntIn, Int)}
        IntIn↑ := Int;
        PUT(IntIn);
        WRITE(Int); {write characters representing
                     integer to OUTPUT}
        READ(Int) {converts INPUT chars to integer}
      END;
    WRITELN
  END.  {GenIntFile}
```

Execution

```
INPUT :1 2 3 4 5 -1
OUTPUT:          1          2          3          4          5
```

The execution does not show the real purpose of GenIntFile, however, which is to create the external file IntIn, whose value following the last WRITE statement is

<<1,2,3,4,5>, <>, W>.

The second part of the program system takes its input from IntIn, and sends output to Out with a copy to the regular output for observation. OddBeforeEven is intended to separate the integers of its input into odds and evens, and list the odd values first, followed by the even values, otherwise preserving the sequence in IntIn.

Design Part 1

```
PROGRAM OddBeforeEven(IntIn, Out, OUTPUT);
  VAR
    IntIn, Out: FILE OF INTEGER;
  BEGIN {OddBeforeEven}
    REWRITE(Out);
    {copy odd members of IntIn to Out};
    {copy even members of IntIn to Out};
    RESET(Out);
    {copy Out to OUTPUT}
  END.  {OddBeforeEven}
```

Since RESET(IntIn) will fill IntIn↑ with the first value in IntIn, this value is copied to Out↑ so that the subsequent PUT(Out) copies the value to Out.

Design Part 1.1

```
{copy odd members of IntIn to Out}
RESET(IntIn);
WHILE NOT EOF(IntIn)
DO
  BEGIN
    IF ODD(IntIn↑)
    THEN
      BEGIN
        {WRITE(Out, IntIn↑)}
        Out↑ := IntIn↑;
        PUT(Out)
      END;
    GET(IntIn)
  END;
```

Design Part 1.2

```
{copy even members of IntIn to Out}
RESET(IntIn);
WHILE NOT EOF(IntIn)
DO
  BEGIN
    IF NOT ODD(IntIn↑)
    THEN
      BEGIN
        {WRITE(Out, IntIn↑)}
        Out↑ := IntIn↑;
        PUT(Out)
      END;
    GET(IntIn)
  END;
```

Design Part 1.3

```
{copy Out to OUTPUT}
WHILE NOT EOF(Out)
DO
  BEGIN
    WRITELN(Out↑);
    GET(Out)
  END
```

Combining these design parts yields the following sample results, using the file IntIn produced by the execution of GenIntFile shown above.

Execution

```
INPUT :
OUTPUT:          1
                 3
                 5
                 2
                 4
```

Many Pascal machines support two other operations on non-TEXT files, allowing programmers to avoid the use of buffer variables. However, these operations are not officially part of the Pascal language, so they may not be available.

```
READ(FileVar, Var1, ..., Varn)
```

where Var1, ..., Varn are variables of the appropriate type, may be defined in terms of GET as follows:

```
READ(FileVar, Var1, ..., Varn)
```

has the effect of:

```
BEGIN
  Var1 := FileVar↑;
  GET(FileVar);
  ...
  Varn := FileVar↑;
  GET(FileVar)
END
```

Similarly,

```
WRITE(FileVar, Exp1, ..., Expn)
```

where Exp1, ..., Expn are expressions of the appropriate type, has the effect of:

```
BEGIN
  FileVar↑ := Exp1;
  PUT(FileVar);
  ...
  FileVar↑ := Expn;
  PUT(FileVar)
END
```

If a Pascal machine supports READ and WRITE on non-TEXT files, OddBeforeEven can be rewritten without using buffer variables:

```
PROGRAM OddBeforeEven(IntIn, Out, OUTPUT);
  VAR
    IntIn, Out: FILE OF INTEGER;
    Val: INTEGER;
  BEGIN
    REWRITE(Out);
    RESET(IntIn);
    WHILE NOT EOF(IntIn)
    DO
      BEGIN {copy odd members of IntIn to Out}
        READ(IntIn, Val);
        IF Odd(Val)
        THEN
          WRITE(Out, Val)
      END;
    RESET(IntIn);
    WHILE NOT EOF(IntIn)
    DO
      BEGIN {copy even members of IntIn to Out}
        READ(IntIn, Val);
        IF NOT Odd(Val)
        THEN
          WRITE(Out, Val)
      END;
    RESET(Out);
    {copy Out to OUTPUT}
    WHILE NOT EOF(Out)
    DO
      BEGIN
        READ(Out, Val);
        WRITELN(Val)
      END
  END.
```

16.2.3 TEXT Files

The meaning of TEXT file operations was given in Sections 6.2.4 and 6.2.5. These definitions agree with those given for non-TEXT files by replacing the lists of elements in the latter with strings, in which the special character / represents a line marker. GET and PUT and buffer variables may thus be used with TEXT files as well. There is one difference involving RESET: when a TEXT file is being written, its final line is incomplete until the line marker is added. If the last operation is WRITELN the marker is present; if not, a RESET adds it. (Many Pascal machines do not properly implement this feature of RESET, which is why in this text we have been careful to always use a final WRITELN.) TEXT files permit the READLN, WRITELN, and EOLN operations that are not available for other files. In addition there is a statement that causes the output to be adjusted to a page boundary if the printing device has such a feature:

```
PAGE(F)
```

causes the next WRITE statement for F to appear at the top of the page.

16.2.4 Justifying Text with InsertExtraBlanks

The existence of INTEGER variables and the use of buffer variables makes complex text processing easier in D Pascal than in CF Pascal. As an extended example, the problem of justifying text will be considered. In this problem, words (defined as sequences of nonblank characters) and blanks alternate in the input lines. A line length L is given as the available output space. The words from input are to be printed in this space, and furthermore, the right and left margins are to be aligned. The most straightforward algorithm for solving the problem is to remove any extra blanks between words, and if the words fit the available space, pad some of the spaces between words to make the margins even.

The procedure RemoveExtraBlanks of Section 10.1.3 will be useful in implementing this text-justifying algorithm. Another useful component will be a routine that begins with a too-short line, and pads it to fill the space L. Suppose there are N words with no extra blanks between them, containing with the blank separators a total of C characters. Then L - C blanks must be fitted into N - 1 slots between words. If there is more than a single word, $N \geq 1$,

$$(L - C) \div (N - 1)$$

blanks can be put in each of the slots. But if the division does not come out even, the remaining R blanks must be distributed. They can be distributed among the rightmost R slots of the line, making those spaces one blank wider than the ones at the beginning of the line. The design based on these ideas is:

Design Part 2

```
PROCEDURE InsertExtraBlanks(VAR FileIn, FileOut: TEXT;
                            LineLength: INTEGER);
  {assumes FileIn contains no extra blanks
     between words;
   copies FileIn to FileOut, uniformly inserting extra
   blanks so that each line in FileOut is LineLength
   characters long}
  VAR
    TFile: TEXT;
    XBlanks: INTEGER;
    Words, Chars: INTEGER;
  BEGIN {InsertExtraBlanks}
    RESET(FileIn);
    REWRITE(FileOut);
    WHILE NOT EOF(FileIn)
    DO
      BEGIN
        {copy one line from FileIn to TFile,
         counting Words and Chars in line};
        XBlanks := LineLength - Chars;
        IF (Words > 1) AND (XBlanks > 0)
        THEN {multiple words on line and
              blanks to insert}
          BEGIN
            {copy line from TFile to FileOut
             inserting blanks}
          END
        ELSE {single word on line, line too long,
              or no blanks to add}
          BEGIN
            {copy entire line from TFile to FileOut,
             inserting any extra blanks at end}
```

```
            END
        END;
      RESET(FileIn);
      RESET(FileOut)
    END;  {InsertExtraBlanks}
```

Since FileIn does not contain extra blanks, each blank or line marker signifies the end of a word.

Design Part 2.1

```
{copy one line from FileIn to TFile,
 counting Words and Chars in line}
REWRITE(TFile);
Chars := 0;
Words := 0;
WHILE NOT EOLN(FileIn)
  DO
    BEGIN
      Chars := Chars + 1;
      IF FileIn↑ = ' '
      THEN
        Words := Words + 1;
      TFile↑ := FileIn↑;
      PUT(TFile);
      GET(FileIn)
    END;
Words := Words + 1;
READLN(FileIn);
WRITELN(TFile);
RESET(TFile);
```

In the next design part, Blanks is the number of extra blanks to insert between two words on the line. All slots between words beyond number Pos get an additional blank.

Design Part 2.2

```
{copy line from TFile to FileOut inserting blanks}
Blanks := XBlanks DIV (Words-1);
Pos := Words - (XBlanks MOD (Words-1));
Word := 0;
WHILE NOT EOF(TFile)
DO
  BEGIN
    {copy word from TFile to FileOut};
    Word := Word + 1;
    IF Word < Words
    THEN {not last word}
      BEGIN
        WRITE(FileOut,' ');
        {insert extra blanks:
         Blanks if Word < Pos, Blanks+1 if Word >= Pos}
      END
    ELSE {last word}
      WRITELN(FileOut)
  END
```

Design Part 2.2.1

```
{copy word from TFile to FileOut}
WHILE TFile↑ <> ' '
DO
  BEGIN
    FileOut↑ := TFile↑;
    PUT(FileOut);
    GET(TFile)
  END;
GET(TFile);
```

Design Part 2.2.2

```
{insert extra blanks:
 Blanks if Word < Pos, Blanks+1 if Word >= Pos}
BIndex := 1;
WHILE BIndex <= Blanks
DO
  BEGIN
    WRITE(FileOut,' ');
    BIndex := BIndex + 1
  END;
IF Word >= Pos
THEN
  WRITE(FileOut,' ')
```

If there is only a single word on the line, all the extra blanks should be inserted to the right of the word. If the value of `XBlanks` is less than or equal to zero, the length of the text line is greater than or equal to `LineLength`, and the entire text is printed.

Design Part 2.3

```
{copy entire line from TFile to FileOut,
 inserting any extra blanks at end}
WHILE NOT EOLN(TFile)
DO
  BEGIN
    FileOut↑ := TFile↑;
    PUT(FileOut);
    GET(TFile)
  END;
WHILE XBlanks > 0
DO
  BEGIN
    WRITE(FileOut,' ');
    XBlanks := XBlanks - 1
  END;
WRITELN(FileOut)
```

These design parts can be assembled into two development programs for testing. The first reports counts of words and characters, and contents of TFile to OUTPUT and prints the value of Words and Chars.

Development Program 2A

```
PROGRAM TestInsert(INPUT, OUTPUT);
  VAR
    TempFile1, TempFile2: TEXT;
    LineLength: 1..65;
    Ch: CHAR;
  {Include
   PROCEDURE RemoveExtraBlanks
     (VAR FileIn, FileOut: TEXT);
     copy FileIn to FileOut removing leading, trailing,
     and extra blanks between words in FileIn}
  PROCEDURE InsertExtraBlanks
    (VAR FileIn, FileOut: TEXT;
     LineLength: INTEGER);
    {assumes FileIn contains no extra blanks
       between words;
     copies FileIn to FileOut, uniformly inserting extra
     blanks so that each line in FileOut is LineLength
     characters long}
    VAR
      TFile: TEXT;
      XBlanks: INTEGER;
      Words, Chars: INTEGER;
    BEGIN {InsertExtraBlanks}
      RESET(FileIn);
      REWRITE(FileOut);
      WHILE NOT EOF(FileIn)
      DO
        BEGIN
          {copy one line from FileIn to TFile,
           counting Words and Chars in line}
          REWRITE(TFile);
          Chars := 0;
          Words := 0;
          WHILE NOT EOLN(FileIn)
          DO
            BEGIN
              Chars := Chars + 1;
              IF FileIn↑ = ' '
              THEN
                Words := Words + 1;
              TFile↑ := FileIn↑;
              PUT(TFile);
              GET(FileIn)
            END;
          Words := Words + 1;
          READLN(FileIn);
          WRITELN(TFile);
          RESET(TFile);
```

```
        {copy TFile to OUTPUT; write Words and Chars}
        WHILE NOT EOF(TFile)
        DO
          BEGIN
            WHILE NOT EOLN(TFile)
            DO
              BEGIN
                OUTPUT↑ := TFile↑;
                PUT(OUTPUT);
                GET(TFile)
              END;
            READLN(TFile);
            WRITELN;
            WRITELN('Words=',Words,', Chars=',Chars)
          END;
        XBlanks := LineLength - Chars;
        IF (Words > 1) AND (XBlanks > 0)
        THEN {multiple words on line and
              blanks to insert}
          BEGIN
            {copy line from TFile to FileOut
             inserting blanks}
          END
        ELSE {single word on line, line too long,
              or no blanks to add}
          BEGIN
            {copy entire line from TFile to FileOut,
             inserting any extra blanks at end}
          END
      END;
    RESET(FileIn);
    RESET(FileOut)
  END;  {InsertExtraBlanks}
BEGIN {TestInsert}
  REWRITE(TempFile1);
  READLN(LineLength);
  WHILE NOT EOF
  DO
    BEGIN
      WHILE NOT EOLN
      DO
        BEGIN
          READ(Ch);
          WRITE(TempFile1,Ch)
        END;
      READLN;
      WRITELN(TempFile1)
    END;
  RemoveExtraBlanks(TempFile1,TempFile2);
  InsertExtraBlanks(TempFile2,TempFile1,LineLength)
END.  {TestInsert}
```

Execution

```
INPUT :15
          Now   is  the time
        for all    good
       men
       to   come to   the
            aid  of   their party.
OUTPUT:Now is the time
       Words          4, Chars         15
       for all good
       Words          3, Chars         12
       men
       Words          1, Chars          3
       to come to the
       Words          4, Chars         14
       aid of their party.
       Words          4, Chars         19
```

The second development program is formed by removing the added code in the last skeleton and adding the remaining design parts and code to copy `TempFile1` to `OUTPUT`. If the same input is presented to this program, the result is:

Execution

```
OUTPUT:Now is the time
       for  all   good
       men
       to come to  the
       aid of their party.
```

Even this small amount of test data covers many of the cases in the program. The first line is exactly the right length; no blanks are added to it. The second and fourth lines each need blanks added, and additional blanks are added before the rightmost words. The third line contains a single word, and the last line is too long.

16.2.5 Interactive Input/Output

To this point, all the programs executed by the Pascal machine have been *batch-processed,* i.e., the programs and all the data are presented to the machine at the same time, the program is executed, and the results are printed. Many programs are designed to be run *interactively,* prompting the user to enter more data after execution has begun and displaying some results before all the data has been entered. `MinMax` below is a typical interactive program. It reads a single character command that guides the program to print either the higher (requested by the command `H`) or lower (requested by `L`) of the two characters following the command. The program continues comparing characters as long as the first character on an input line is a valid command. When an invalid command is encountered, the program terminates.

```
PROGRAM MinMax(INPUT,OUTPUT);
  VAR
    Command,Ch1, Ch2: CHAR;
  PROCEDURE SortTwo(VAR Ch1, Ch2: CHAR);
  { <Ch1,Ch2> := <min(Ch1,Ch2),max(Ch1,Ch2)> }
    VAR
      Temp: CHAR;
    BEGIN {SortTwo}
      IF Ch1 > Ch2
      THEN
        BEGIN
          Temp := Ch1;
          Ch1 := Ch2;
          Ch2 := Temp
        END
    END;  {SortTwo}
  BEGIN {MinMax}
    WRITELN('Enter <H or L> <ch1> <ch2>');
    READ(Command);
    WHILE (Command = 'H') OR (Command = 'L')
    DO
      BEGIN
        READLN(Ch1,Ch2);
        SortTwo(Ch1,Ch2);
        IF Command = 'H'
        THEN
          WRITELN('The greater character is ''',
                  Ch2,'''')
        ELSE
          WRITELN('The lesser character is ''',
                  Ch1,'''');
        READ(Command)
      END
  END. {MinMax}
```

Execution

```
OUTPUT:Enter <H or L> <ch1> <ch2>
INPUT :HFG
OUTPUT:The greater character is 'G'
INPUT :LFG
OUTPUT:The lesser character is 'F'
INPUT :J
OUTPUT:
```

As is typical of interactive program, MinMax prompts its user for input with the line:

```
Enter <H or L> <ch1> <ch2>
```

and only a user unacquainted with BNF would think that the angle brackets should be entered in response. To be useful this line must appear immediately, but there is a peculiarity of Pascal that may hold it until some input has been entered. Program execution begins with an implicit RESET(INPUT);REWRITE(OUTPUT). The RESET does a GET(INPUT) to place the first character of the file in INPUT↑, but no data has been entered because the user is waiting for the prompt! Furthermore, when no data has been entered what is the value of EOF or EOLN? Pascal machines usually get around these problems by defining default values for INPUT↑, EOF, and EOLN, or using *lazy evaluation* strategies. The programs in this text assume lazy evaluation.

The default values that are typically used are:

Quantity	Initial Value
EOF(INPUT)	FALSE
INPUT↑	□
EOLN(INPUT)	TRUE

With these values, a program's first READ statement obtains a blank, and INPUT↑ assumes the value of the first character actually in the file. Thus when a Pascal machine adopts this method, the programmer must do an initial GET or READLN to discard the first character.

Lazy evaluation is a strategy by which values are not computed until they are needed. Since no value associated with INPUT is needed in MinMax until the first READ statement (READ(Command)) is encountered, execution of the implicit RESET(INPUT) is postponed until after the first WRITELN statement.

In either case, an initial WRITE statement can provide a prompt for an interactive program. However, if lazy evaluation is used, no extra READ statement is required, and using one will discard the first real character of input. So experimentation with the Pascal machine is required before interactive programs can be written.

16.2.6 Exercises

16.2.1 TEXT files may be examined using either READ statements or through the buffer variable. Give a typical program loop that uses a READ statement to obtain each element in a TEXT file, ignoring the line structure. Then write an equivalent program using the buffer-variable method of access.

16.2.2 The OddBeforeEven program makes two passes over the file IntIn. Redo the entire design so that only one pass is required, using a temporary file if required. Instead of using Out, send the results directly to OUTPUT.

16.2.3 Design and test a programming system using two external files, one a FILE OF Month and the other a FILE OF INTEGER. One program in the system writes these files, putting in the month and number of inches of rainfall for that month. (Simply assume that each month a separate program is made up to write that month's data, and give the program for a February where the rainfall was 3 inches.) The second program summarizes a year's data and prints out a table; this program is to be run after the December data is entered. Is a third program needed to initialize the files?

16.2.4 The following declarations are in force for this question:

```
TYPE
  Drink = (Coffee, Tea, Milk);
VAR
  Ch: CHAR;
  Itg: 3..33;
  Enm: Drink;
  EFile: FILE OF Drink;
FUNCTION Ff(VAR Pr1: CHAR; Pr2: CHAR): BOOLEAN;
```

In each part below there is at least one mistake. Locate and explain each one.

a)
```
IF Ff(Ch, Ch) = 1
  THEN
    EFile↑ := SUCC(Tea)
```

b) (within the body of Ff):
```
Ff := Ch IN ['A', Enm]
```

c)
```
IF (Coffee < Tea) AND (Ff('A', 'B'))
    THEN
      Itg := Itg + 32
```

d)
```
IF [3, 44, Itg] + [Tea, PRED(Tea)] = []
    THEN
      RESET(EFile)
```

16.2.5 Extend the text-justifying problem of Section 16.2.4 to include rearranging the words on lines so that no line is too long. That is, the words are to be taken from `INPUT`, and justified lines are to be sent to `OUTPUT`, with enough words to fill, but not overfill each line. The most straightforward algorithm is to read words until the sum of their lengths (including one-blank separators) just exceeds the output length available, then justify the line using all but the final word, which is taken to start the next line. Give a program design that solves the extended problem, using as many of the routines from Section 16.2.4 as possible.

16.2.6 The simple algorithm suggested in the previous exercise does not always produce the best looking text.

a) Describe how simple hyphenation could be added to the algorithm, where only the following suffixes are broken off by hyphen:

-tion
-ing
-ed

b) Describe how the algorithm could be modified to hold three lines, and minimize the number of spaces that are inserted by moving words between lines. Give an example in which the three-line algorithm would produce text with a better appearance.

16.2.7 Investigate the use of prompt lines in interactive programs on your Pascal machine.

a) Is an initial discarded READ statement needed?

b) Explain how a program expecting free-form input in which all leading blanks are of no significance, can be written with a prompt line so that it works on Pascal machines, whether they use a default value for the buffer variable or lazy evaluation.

16.3 Records

Preview

When a complex data object is to be used in a program, it is convenient to have a single name for it, and to declare it without explicitly repeating all the names of its parts. The Pascal record types allow the object's description to be given once, then used either by its composite name, or by naming its separate parts.

Records permit a fixed number of related data items of (possibly) different types to be grouped together in a single object. Each data item, called a *field*, consists of a name, a colon, and a type. In the following example, there are two record types, `Date` and `WorkerInfo`, each containing three fields. Some of the fields of the `WorkerInfo` type are themselves records. Thus, record types can be nested to create descriptions of arbitrarily complex objects.

```
TYPE
  Month = (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);
  DayNum = 1 .. 31;
  ValidYear = 1850 .. 2001;
  Date = RECORD
           Mo: Month;
           Day: DayNum;
           Year: ValidYear
         END;
  WorkerInfo = RECORD
                 SSNumber: INTEGER;
                 BirthDay: Date;
                 FirstHired: Date
               END;
```

When a variable has been declared to be of a record type, its name refers to the entire record. As a summary name, the variable is useful, for example when the entire record is to be passed as an argument to a procedure. However, to manipulate the parts of a record, other names are needed. The fields are named by joining their identifiers (from the type declaration) to the variable name with a period. For example, the variable

```
VAR
  Worker: WorkerInfo;
```

declares `Worker` to be a record with the three fields named in the type declaration above as `SSNumber`, `BirthDay`, and `FirstHired`. The names of the parts of this particular `WorkerInfo` are

```
Worker.SSNumber
Worker.BirthDay
Worker.FirstHired
```

where the first is an elementary item, and could be given a value:

```
Worker.SSNumber := 123456789
```

The other two parts of `Worker` are themselves records, and their parts must be further identified. For example:

```
Worker.BirthDay.Mo := Feb;
Worker.FirstHired.Day := 22;
```

The values of record variables encompass all the possible values of their components, in all possible combinations. Some of these combinations may not correspond to the idea that the programmer had in mind for the type. For example, the author of the type `Date` above has made an attempt to restrict each field to sensible values, but nothing prevents the construction of dates like February 29, 1961, or September 31, 1901 in which there is a mismatch between the components.

Within a record, field names must be unique. However, since references to fields must contain the names of all records of which the field is a component, fields in inner records can be named by the same identifiers as fields in outer records or other variables. No ambiguity arises from this reuse of identifiers. Thus the following Pascal fragment is legal:

```
VAR
  X: INTEGER;
  Y: RECORD
       X: RECORD
            X: INTEGER
          END
     END;
BEGIN
  ...;
  WRITELN(X, Y.X.X)
END
```

The innermost integer field named X is named by prefixing the names of all records that contain it (that is, Y and X).

A composite record name, the record variable (as distinguished from its fields), may not be assigned anything but another record variable of the same type, and that only if none of the fields are files. However, a record variable may be passed as an actual VAR parameter. The procedure to which it is passed may then alter its fields, so that the effect in the calling program is to alter the record through its composite name.

16.3.1 Record Syntax and Meaning

The syntax rules for records appear below:

<unpacked structured type> ::= SET OF <base type>
  ¦ FILE OF <component type> ¦ RECORD <field list> END

<field list> ::= <fixed part>

<fixed part> ::= <record sections>

<record sections> ::= <record sections> ; <record section>
  ¦ <record section>

<record section> ::= <identifier list> : <type denoter>

The syntax rules for <variable access> must also be changed since sequences of identifiers separated by periods are now permitted.

<variable access> ::= <variable identifier> ¦ <buffer variable>
  ¦ <component variable>

<buffer variable> ::= <file variable> ↑

<file variable> ::= <variable access>

<component variable> ::= <field designator>

<field designator> ::= <record variable> . <field specifier>
  ¦ <field designator identifier>

<field designator identifier> ::= <identifier>

<record variable> ::= <variable access>

<field specifier> ::= <identifier>

Because the components of record variables are objects of previously described types, and because each component has a name that is a series of identifiers separated by periods, records can be incorporated into the program calculus by simply placing these identifiers and their associated values in execution states.

16.3.2 Sorting Dates

When reading or writing several collections of related values to files, it is possible to use a separate file for each collection. For example, to record a collection of dates, the Month of each could be written to one file, the Day to another, and the Year to a third. These files would then be, respectively:

```
FILE OF Month
FILE OF DayNum
FILE OF ValidYear
```

The dates would be kept together by position in the three files: the first items written would be one date, the second another, etc. A much easier way to handle composite items is to group the related values in a single record and transmit the record value with a single read or write.

The following program reads a list of dates from INPUT (one date, month and day, per line), sorts the dates into ascending order, and prints the list. Each date has the following form:

<date> ::= <month> <day integer>

<month> ::= JAN ¦ FEB ¦ MAR ¦ APR ¦ MAY ¦ JUN
¦ JUL ¦ AUG ¦ SEP ¦ OCT ¦ NOV ¦ DEC

where <day integer> lies between 1 and 31.

Instead of developing the sorting program directly, it is wise to first define some data types to make the sorting task easier. Routines will be needed to read and write dates, and to compare one date to another. In Section 14.1.2, procedures were developed to create a type Month whose values could be read from INPUT, written to OUTPUT, and compared to one another, as follows:

```
TYPE
  Month = (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);

PROCEDURE ReadMonth(VAR FIn: TEXT; VAR Mo: Month);
  {FIn.3=R and length(FIn.2)>=3  -->
     read three characters from FIn.2 and assign the
     corresponding Month value to Mo if possible;
     otherwise assign NoMonth to Mo}

PROCEDURE WriteMonth(VAR FOut: TEXT; VAR Mo: Month);
  {FOut.3=W and Mo<>NoMonth  -->
     write the three characters corresponding to Mo's
     value to FOut.1}
```

Since objects of type Month are represented as constants from an enumerated type, no comparison function needs to be written to determine if one Month value is less than another.

A similar type could be developed for day of the month, starting with the type:

```
TYPE
  DayNum = 1..31
```

However, no routines need be written to read, write, or compare DayNum values, since Pascal permits the direct operations on INTEGER, of which DayNum is a subrange type.

Since a date is just a month and a day, it can be represented as a record and its operations can be built with the more primitive operations for Month and DayNum.

```
TYPE
  Date = RECORD
           Mo: Month;
           Day: DayNum
         END;
{Include PROCEDURE ReadMonth(VAR FIn: TEXT; VAR Mo: Month)}
PROCEDURE ReadDate (VAR FIn: TEXT; VAR Result: Date);
  {FIn.3=R and length(FIn.2)>=5  -->
     read three characters representing a Month and
     an integer representing a day from FIn.2, and
     assign the corresponding Month and DayNum values
     to Result.Mo and Result.Day}
  BEGIN {ReadDate}
    ReadMonth(FIn,Result.Mo);
    READ(FIn,Result.Day)
  END; {ReadDate}
{Include PROCEDURE WriteMonth(VAR FOut: TEXT; VAR Mo: Month)}
PROCEDURE WriteDate(VAR FOut: TEXT; VAR D: Date);
  {FOut.3=W and D.Mo<>NoMonth  -->
     write the character representation corresponding
     to the values of D.Mo and D.Day to FOut.1}
  BEGIN {WriteDate}
    WriteMonth(FOut,D.Mo);
    WRITE(FOut,D.Day:3)
  END;  {WriteDate}
FUNCTION Less(VAR D1, D2: Date): BOOLEAN;
  { Less := D1 < D2 }
  BEGIN {Less}
    IF D1.Mo < D2.Mo
    THEN
      Less := TRUE
    ELSE
      IF D1.Mo > D2.Mo
      THEN
        Less := FALSE
      ELSE {D1.Mo = D2.Mo}
        Less := (D1.Day < D2.Day)
  END;  {Less}
```

Although `Less` is a function that does not modify its parameters, they are declared as variable parameters rather than value parameters. Value parameters require a copy operation (hidden by the Pascal machine) to ensure that changes within the function cannot reach the outside; this overhead is not present with variable parameters. Some Pascal machines do not permit the use of record types in value parameters for this reason.

In these data types, the type and procedure declarations have been kept in the same unit of scope, to emphasize their connection. This method of organization produces a "flat" procedure structure—all the procedures are declared in the same unit of scope and (avoiding the restriction of declaration before use) any procedure can be called by any other procedure. A different, more nested program structure could also be adopted based on procedure-call relationships. For example, since `ReadMonth` is called only by `ReadDate`, the declaration for the former could be nested within the declaration of the latter. A similar relationship holds between `WriteMonth` and `WriteDate`. Thus, the following program structure could be used:

```
PROCEDURE ReadDate (VAR FIn: TEXT; VAR Result: Date);
  PROCEDURE ReadMonth(VAR FIn: TEXT; VAR Mo: Month);
    ...;
  BEGIN {ReadDate}
    ...
  END;  {ReadDate}

PROCEDURE WriteDate(VAR FOut: TEXT; VAR D: Date);
  PROCEDURE WriteMonth(VAR FOut: TEXT; VAR Mo: Month);
    ...;
  BEGIN {WriteDate}
    ...
  END;  {WriteDate}

FUNCTION Less(VAR D1, D2: Date): BOOLEAN;
  BEGIN {Less}
    ...
  END;  {Less}
```

With this organization, only `ReadDate` can call `ReadMonth`; only `WriteDate` can call `WriteMonth`. Restricting the availability of procedures hides information. Whether the flat structure emphasizing the relationship between type definitions and operations is better than the nested structure hiding names is a matter of taste.

The program to sort dates can be developed with either of the organizations above, and most of the work is already accomplished by the existence of the right data types. A single date is read and inserted in a file of dates (`DateFile`). Subsequent dates are inserted in the file so that it remains in ascending order.

Design Part 3

```
PROGRAM SortDates(INPUT,OUTPUT);
  TYPE
    Month = (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
             Jul, Aug, Sep, Oct, Nov, Dec);
    DayNum = 1..31;
    Date = RECORD
             Mo: Month;
             Day: DayNum
           END;
    FileOfDate = FILE OF Date;
  VAR
    DateFile: FileOfDate;
  {Include ReadDate, WriteDate, Less, ReadMonth, WriteMonth}
  BEGIN {SortDates}
    REWRITE(DateFile);
    ReadDate(INPUT,DateFile↑);
    READLN; {Skip over line marker between dates}
    PUT(DateFile);
    WHILE NOT EOF
    DO
      {insert a new date in ascending order
       in DateFile};
    {copy DateFile to OUTPUT}
  END.  {SortDates}
```

To keep the values in `DateFile` in ascending order, all those dates smaller than a new date `D` are copied from `DateFile` to a temporary file `TFile`, `D` is written to `TFile` followed by the

remaining dates in DateFile, and TFile is copied back into DateFile. (Declarations for D and TFile must be added to Design Part 3.)

Design Part 3.1

```
{insert a new date in ascending order in DateFile}
BEGIN
  ReadDate(INPUT,D);
  READLN; {Skip over line marker between dates}
  {copy elements smaller than D
   from DateFile to TFile};
  TFile↑ := D;
  PUT(TFile);
  {copy remainder of DateFile to TFile};
  {copy TFile to DateFile}
END
```

Dates are copied from DateFile to TFile until either no more dates remain in the former or until a date is greater than the value of D (i.e., Less(DateFile↑,D) has the value FALSE). A declaration for the Boolean variable Copying must be added to Design Part 1.

Design Part 3.1.1

```
{copy elements smaller than D from DateFile to TFile}
BEGIN
  RESET(DateFile);
  REWRITE(TFile);
  Copying := TRUE;
  WHILE NOT EOF(DateFile) AND Copying
  DO
    IF Less(DateFile↑,D)
    THEN
      BEGIN
        TFile↑ := DateFile↑;
        PUT(TFile);
        GET(DateFile)
      END
    ELSE
      Copying := FALSE
END
```

The three copying operations are all straightforward. When DateFile is copied to OUTPUT, only one date is written on a line.

Design Part 3.1.2

```
{copy remainder of DateFile to TFile}
WHILE NOT EOF(DateFile)
DO
  BEGIN
    TFile↑ := DateFile↑;
    PUT(TFile);
    GET(DateFile)
  END;
```

Design Part 3.1.3

```
{copy TFile to DateFile}
RESET(TFile);
REWRITE(DateFile);
WHILE NOT EOF(TFile)
DO
  BEGIN
    DateFile↑ := TFile↑;
    PUT(DateFile);
    GET(TFile)
  END
```

Design Part 3.2

```
{copy DateFile to OUTPUT}
RESET(DateFile);
WHILE NOT EOF(DateFile)
DO
  BEGIN
    WriteDate(OUTPUT,DateFile↑);
    WRITELN;
    GET(DateFile)
  END
```

A sample execution of the assembled program is shown below.

Execution

```
INPUT :FEB 1
       JAN29
       JAN30
       DEC25
OUTPUT:JAN 29
       JAN 30
       FEB  1
       DEC 25
```

16.3.3 Exercises

16.3.1 Suppose that a file has record components, and one component of the record is a file. Give such a declaration, and give the form of the buffer variable of the inner file.

16.3.2 Consider the following declarations:

```
CONST
  Upper=19;
TYPE
  Material=(Brick, Block, Stone, Lumber);
VAR
  St: SET OF Material;
  Rd: RECORD
        A: INTEGER;
        B: Material;
        C: Upper..Upper
      END
```

a) Give an execution state whose contents goes with these declarations, including an appropriate value for each of the objects declared (not "?" but some value that might legally be assigned). For the remaining parts of this problem, call your state S.

b) Calculate the value `Rd.B IN St` (S).

c) Calculate the value `St := St + [Block .. Lumber] + [] + [Rd.B]` (S), showing each step.

16.4 WITH Statement

Preview

Pascal provides a convenient way to avoid writing the outer fields of a nested record description.

Records can be composed to achieve complex types. Consider the following declarations:

```
TYPE
  Month = (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);
  Date = RECORD
           Mo: Month;
           Day: 1 .. 31;
           Year: INTEGER
         END;
  PRecord = RECORD
              ...;
              BirthDay: Date;
              ...
            END;
VAR
  Smith: PRecord;
```

Each `Date` has three selectors: `Mo`, `Day`, and `Year`. `PRecords` associate dates with personnel. If a `BirthDay` were to be assigned to `Smith`, the sequence of assignments might be:

```
Smith.BirthDay.Mo := Feb;
Smith.BirthDay.Day := 28;
Smith.BirthDay.Year := 1960
```

Repeating the names of the fields in each statement is tedious and error-prone. The WITH statement allows references like this to be factored into a statement heading. For example, the previous assignments can be written:

```
WITH Smith.BirthDay
DO
  BEGIN
    Mo := Feb;
    Day := 28;
    Year := 1960
  END
```

The syntax rules for WITH statements appear below.

<with statement> ::= WITH <record variable list> DO <statement>

<record variable list> ::= <record variable list> , <record variable>
| <record variable>

<record variable> ::= <variable access>

16.4.1 Exercises

16.4.1 Suppose the example above had begun:

```
WITH Smith
DO
```

Draw a syntax tree to show that this is legal, and write the appropriate assignment statements.

16.4.2 Give a context rule to accompany the WITH statement syntax, that captures the idea that the header must "go with" the declarations and the statement following, and which would forbid (among others):

```
WITH Smith.Day
DO
  Birthday := 29
```

16.5 Summary

Declarations for the aggregate data types SET, FILE, and RECORD take the form:

```
SET OF T
FILE OF T
RECORD
  F1: T1;
  ...;
  Fn: Tn
END
```

Where T, T_1, ... , T_n are types and F_1, ..., F_n are selector identifiers. Sets and files have components of a single type; records may have components with different types. Set components must have ordinal types other than INTEGER, although limitations on the number of components are imposed by each Pascal machine. File components may be any type except another file or a record containing a file component. Records have a fixed number of components specified by the declaration. Set values may consist of zero or more components, but a maximum number is fixed by the declaration. Files may grow to arbitrary lengths.

The members of a set cannot be explicitly selected, but the IN operator can test for the presence of a given member. Only a single component of a file F can be accessed at any time during execution by referring to the file's buffer variable F↑. Any component of a record can be selected for reference or assignment by appending a period and the name of the component to a variable of the appropriate type.

When choosing between the value and variable transmission mechanism for parameters with aggregate types, considerations of security must be balanced against those of efficiency. Value parameters are more secure because inadvertent assignments to them do not change the execution state where the call was made. However, value parameters require that separate copies of their values are made in the program state of the called procedure each time the procedure is called. This can be quite expensive for aggregate types. Thus, ordinal-type values are passed as value parameters unless the called procedure uses them to return one of several results, but aggregate values are often passed as variable parameters even if they are not updated by the procedure.

CHAPTER 17

DESIGNING SEQUENTIAL CONTROL

Chapter Preview

The control statements described in this chapter are not essential for programming. The effect of each can be obtained using a combination of statements already described. However, each statement is convenient in common programming situations, and the programmer who has them handy often finds them a natural choice.

The CASE, FOR, and REPEAT statements of this chapter permit programmers to express their intentions more precisely and more conveniently than with the control statements of CF Pascal. Where the IF statement selects one of two statements for execution, the CASE statement selects one of what may be many alternatives. Like the WHILE statement, FOR and REPEAT statements control iteration. In a FOR statement, the control is by a count, so FOR statements are natural when the necessary number of repeated executions is known. In contrast, the WHILE statement is appropriate when the number of iterations is not known, but a test condition for stopping the repetition is available. WHILE iterations may be unbounded, but FOR iterations may not. The REPEAT statement also controls an unbounded iteration, but where the WHILE test for termination occurs before each iteration, and is cast in the form of a condition required to continue, the REPEAT test occurs after each iteration, and is in the form of a condition required to stop.

17.1 The CASE Statement

Preview

It is common to use nested IF statements to select a single statement for execution from many. The CASE statement can often replace such a nested IF with a gain in clarity and efficiency.

A CASE statement is formed from an ordinal-type selection expression and a list of statements, each statement labeled by one or more constants of the same ordinal type. For example, in

```
CASE A > B OF
  TRUE:  Max := A;
  FALSE: Max := B
END
```

the selection expression is `A > B` of type `BOOLEAN` and the labels are the `BOOLEAN` constants `TRUE` and `FALSE`. The selection expression is evaluated and the statement labeled with that value (if any) is selected for execution. After the selected statement has finished execution, control is transferred to the statement following the CASE statement. No constant label may be repeated, so at most one statement can be executed. If no statement carries a label with the value of the selection expression, the result of executing the CASE statement is undefined.

A CASE statement can be nearly simulated with nested IF statements. Consider the CASE statement:

```
CASE Exp OF
  L1a, L1b: S1;
  L2:       S2;
  ...;
  LN:       SN
END
```

The IF statement:

```
IF (Exp = L1a) OR (Exp = L1b)
THEN
  S1
ELSE
  IF Exp = L2
  THEN
    S2
  ELSE
      ...
        ELSE
          IF Exp = LN
          THEN
            SN
```

has the same effect when the value of Exp is one of L1a, L1b, L2, ..., LN. However, when the value of Exp is not listed as a label, this IF statement acts as a null statement, but the CASE statement is undefined. (Many Pascal machines implement the CASE statement incorrectly, making it act like the IF statement for a missing label.)

The use of the CASE statement reduces the depth of the statement nesting and the need for logical operators, making a program easier to read. The CASE statement is also more efficient than the IF statement because the CASE selection expression is evaluated only once instead of for each comparison.

17.1.1 CASE Statement Examples

CASE statements are particularly useful with enumerated types. The procedure WriteMonth writes a string value corresponding to an enumerated-type value of the type Month.

```
PROCEDURE WriteMonth(VAR FOut: TEXT; Mo: Month);
  {FOut.3=W and Mo<>NoMonth  -->
     write the three characters corresponding to Mo's
     value to FOut.1}
  BEGIN {WriteMonth}
    CASE Mo OF
      Jan: WRITE(FOut,'Jan');
      Feb: WRITE(FOut,'Feb');
      Mar: WRITE(FOut,'Mar');
      Apr: WRITE(FOut,'Apr');
      May: WRITE(FOut,'May');
      Jun: WRITE(FOut,'Jun');
      Jul: WRITE(FOut,'Jul');
      Aug: WRITE(FOut,'Aug');
      Sep: WRITE(FOut,'Sep');
      Oct: WRITE(FOut,'Oct');
      Nov: WRITE(FOut,'Nov');
      Dec: WRITE(FOut,'Dec')
    END
  END;  {WriteMonth}
```

WriteMonth assumes that Mo has been set to one of the constants of type Month, and explicitly

excludes NoMonth from its domain. A more robust version of WriteMonth could check that Mo had one of the twelve Month values before executing the CASE statement to print the string corresponding to the value.

```
IF Mo IN [Jan..Dec]
THEN
  CASE Mo OF
    Jan: WRITE(FOut,'Jan');
    Feb: WRITE(FOut,'Feb');
    Mar: WRITE(FOut,'Mar');
    Apr: WRITE(FOut,'Apr');
    May: WRITE(FOut,'May');
    Jun: WRITE(FOut,'Jun');
    Jul: WRITE(FOut,'Jul');
    Aug: WRITE(FOut,'Aug');
    Sep: WRITE(FOut,'Sep');
    Oct: WRITE(FOut,'Oct');
    Nov: WRITE(FOut,'Nov');
    Dec: WRITE(FOut,'Dec')
  END
```

By protecting the CASE statement in this way, the programmer can be sure that it will never abort.

CASE statements are also useful with data types containing many values, if the cases to be processed are only a small collection. For example, a routine that reads characters, and takes special action for certain characters, can be written in the pattern:

```
WHILE NOT EOF
DO
  BEGIN
    READ(Ch);
    IF Ch IN ['+', '-', '*', '/']
    THEN
      CASE Ch OF
        '+', '-':
          BEGIN
            WRITELN('Adding operator encountered.');
            CASE Ch OF
              '+': NumPlus := NumPlus + 1;
              '-': NumMinus := NumMinus + 1
            END
          END;
        '*', '/': WRITELN('Multiply/Divide encountered.')
      END
    ELSE
      NumOther := NumOther + 1
  END
```

Notice the nested CASE statement in which actions are grouped and then split again.

Similarly, a few cases in an infinite type can be handled conveniently:

```
IF AnswerCount <= 1
THEN
  CASE AnswerCount OF
    0: WRITE('There are none.');
    1: WRITE('There is 1.')
  END
ELSE
  WRITE('There are ', AnswerCount:3)
```

17.1.2 CASE Statement Definition

The syntax rules for the CASE statement appear below.

<case statement> ::= CASE <expression> OF <case list> END

<case list> ::= <case list> ; <case> | <case>

<case> ::= <constant list> : <statement>

<constant list> ::= <constant list> , <constant> | <constant>

The CASE statement, like the IF statement, can be given a meaning as a union of functions. Let E be an expression yielding an ordinal-type value, and C_1, C_2, and C_3 distinct constants of the same type, and S_1 and S_2 statements. Then the CASE statement S:

```
CASE E OF
  C1, C2: S1;
  C3: S2
END
```

has meaning

$$[\![S]\!] = \{<u, [\![S_1]\!](u)>: [\![E]\!](u) = [\![C_1]\!](u) \text{ OR } [\![E]\!](u) = [\![C_2]\!](u)\}$$
$$\cup \{<u, [\![S_2]\!](u)>: [\![E]\!](u) = [\![C_3]\!](u)\}.$$

The generalization to more labels and more statements is evident. Note that if none of the selectors has the value of E, the function $[\![S]\!]$ is undefined.

17.1.3 Exercises

17.1.1 Explain how to construct a CASE statement that exactly simulates the effect of any given IF statement.

17.1.2 The following declarations are in force for this question:

```
TYPE
  Drink = (Coffee, Tea, Milk);
VAR
  Ch: CHAR; Itg: 3..33;
  Enm: Drink;
  EFile: FILE OF Drink;
```

In each part below there is at least one mistake. Locate and explain each one.

a)
```
CASE SUCC(Tea) OF
  Enm: Ch := 'X';
  Coffee: Ch := 'Y'
END
```

b)
```
CASE Itg OF
   'Y': Ch := 'X';
    X: Ch := 'Y'
END
```

c)
```
CASE SUCC(Itg) OF
   Tea: Ch := 'X';
   Coffee: Tea: Ch := 'Y'
END
```

17.1.3 In the skeleton CASE statements below, either give as much information about the type of variable `Sel` as you can, or explain why the statement is syntactically incorrect:

a)
```
CASE Sel OF
   'X': ...;
   'Z': ...
END
```

b)
```
CASE PRED(Sel) OF
   X: ...;
   Z: ...
END
```

17.1.4 Explain why the following pattern is not a useful one for the situation in which two cases have a common action, and one has an additional action:

```
CASE Ch OF
  'A', 'B':
     BEGIN
       {Common action for 'A' and 'B'};
       CASE 'A' OF
         'A':  {Additional action for 'A'}
       END
     END;
  'C': {Other cases...}
END
```

17.1.5 It has been proposed that the constants used in CASE statements to label each component statement should be generalized to allow any expression as a label. Explain informally what the meaning of such a generalized CASE statement should be, and as a part of your explanation, criticize the proposal.

17.2 The FOR Statement

Preview

Iteration control in a FOR statement is by stepping through the values of some ordinal type, and the current step value is available for calculation within the statement.

Let V be a variable of some ordinal type, and A and Z be expressions of the same type. The FOR statement:

```
FOR V := A TO Z
DO
  S
```

executes the statement S a number of times defined by the values of A and Z before the iteration begins. Suppose that the value of A is p and the value of B is q at the outset. Then the first time S is executed the value of V is p, the second time, V takes value `SUCC(`p`)`, the third time,

SUCC(SUCC(*p*)), and so on until the value for *V* reaches *q* in succession, which is the last execution of *S*. Thus the number of executions is *q* - *p* + 1, so long as this value is at least 1; if not, that is if *p* > *q*, *S* is not executed at all. A FOR statement may also be written to iterate the same number of times, but in the reverse order:

```
FOR V := Z DOWNTO A
DO
  S
```

where now the values taken by *V* are successively

q, PRED(*q*), PRED(PRED(*q*)), ..., *p*.

V is called the *control variable* of the FOR statement. At the end of the iteration, the value of *V* is undefined.

17.2.1 Examples of FOR Statements

Because the control variable takes on successive values of its ordinal type on successive iterations, the FOR statement is ideal for controlling an action that must be repeated over a collection. For example, the enumerated type Month can be stepped through using

```
FOR Mo := Jan TO Dec
DO
  ...
```

more naturally than with the WHILE statement of PrintDays in Section 15.5.2.

The INTEGER type is also a natural one to control with a FOR statement. The following program might be the skeleton for producing student average grades; it reads a set of scores and averages them.

```
PROGRAM AverageScore(INPUT,OUTPUT);
  CONST
    NumberOfScores = 5;
    ClassSize = 3;
  TYPE
    Score = 0..100;
  VAR
    WhichScore: 1..NumberOfScores;
    Student: 1..ClassSize;
    NextScore: Score;
    Ave, TotalScore, ClassTotal: INTEGER;
  BEGIN {AverageScore}
    ClassTotal := 0;
    WRITELN('Student averages:');
    FOR Student := 1 TO ClassSize
    DO
      BEGIN
        TotalScore := 0;
        FOR WhichScore := 1 TO NumberOfScores
        DO
          BEGIN
            READ(NextScore);
            TotalScore := TotalScore + NextScore
          END;
        READLN;
        TotalScore := TotalScore*10;
        Ave := TotalScore DIV NumberOfScores;
        IF Ave MOD 10 >= 5
        THEN
```

```
        WRITE (Ave DIV 10 + 1)
      ELSE
        WRITE(Ave DIV 10);
      ClassTotal := ClassTotal + TotalScore;
    END;
  WRITELN;
  WRITELN('Class average:');
  ClassTotal := ClassTotal DIV (ClassSize*NumberOfScores);
  WRITELN(ClassTotal DIV 10, '.', ClassTotal MOD 10:1)
END.
```

Execution

```
INPUT :90 100 85 85 80
       90 80 75 90 90
       0 50 80 85 80
OUTPUT:Student averages:
              88        85        59
       Class average:
              77.3
```

This program expects individual scores on a line, and a line for each person. The number of scores/line, and the number of lines, are given by the constant definitions. Individual averages are rounded to the nearest integer; the overall average is given to one decimal place (truncated).

17.2.2 FOR Statement Definition

The syntax rules for the FOR statement are:

<for statement> ::= FOR <variable identifier> :=
    <for list> DO <statement>

<for list> ::= <expression> TO <expression>
    | <expression> DOWNTO <expression>

The control variable and the expressions in the <for list> may be any ordinal type. The control variable must be an identifier and may not appear on the left side of an assignment statement within the loop. The control variable's value is undefined at the end of execution of the FOR statement. On a particular Pascal machine the control variable may have a value when the FOR statement is finished, but it is poor programming practice to use this value in further computations because it may or may not be the same for other Pascal machines.

The meaning of a FOR statement can be given by an equivalent WHILE statement, but care must to taken to account for several differences between the two statements. The most straightforward correspondence is wrong. The FOR statement:

```
FOR Index := E1 TO E2
DO
  S
```

is not equivalent to:

```
Index := E1;
WHILE Index <= E2
DO
  BEGIN
    S;
    Index := SUCC(Index)
  END
```

The control variable in the WHILE statement is not undefined as it should be for the FOR statement. The body of the WHILE statement will be executed for the last time when Index

already has the same value as E2. If this value were the last in an enumerated type, the final evaluation of SUCC in the body would abort, where the FOR statement would not abort. Also, assignment statements in the collection of statement S whose targets are variables in E2 would change the number of iterations in the WHILE statement, but not in the FOR statement. The FOR statement below iterates five times, even though First and Last change after execution begins:

```
First := 1;
Last := 5;
FOR Index := First TO Last
DO
  BEGIN
    First := First + 1;
    Last := Last - 1;
    WRITELN(Index, First, Last)
  END
```

Execution

```
OUTPUT:          1        2        4
                 2        3        3
                 3        4        2
                 4        5        1
                 5        6        0
```

Some of these differences can be removed by using control variables that occur nowhere else in the program (say T1 and T2), and by unrolling the first execution of the loop:

```
T1 := E1;
T2 := E2;
IF T1 <= T2
THEN
  BEGIN
    Index := T1;
    S;
    WHILE Index <> T2
    DO
      BEGIN
        Index := SUCC(Index);
        S
      END
  END
```

Although this sequence of statements is closer to the meaning of the FOR statement, it still leaves Index defined at the end of execution.

A direct definition of the FOR statement can be given as follows:

FOR V := E1 TO E2 DO S

where V is a variable of ordinal type T, has the meaning:

$$\boxed{\texttt{FOR V := E1 TO E2 DO } S}(t) =$$

$$(\boxed{\texttt{VAR V: } T}^{T} \circ \boxed{\texttt{VAR V: } T})(t_n) \text{ if } \boxed{\texttt{SUCC}}^{n-1}(\boxed{\texttt{E1}})(t) = \boxed{\texttt{E2}}(t), \text{ where}$$

$$t_n = \boxed{\texttt{S}}(\boxed{\texttt{V := SUCC(V)}}(t_{n-1}))$$

$$\ldots$$

$$t_1 = \boxed{\texttt{S}}(\boxed{\texttt{V := E1}}(t));$$

$$(\boxed{\texttt{VAR V: } T}^{T} \circ \boxed{\texttt{VAR V: } T})(t) \text{ if } \boxed{\texttt{E1}}(t) > \boxed{\texttt{E2}}(t).$$

By composing $\boxed{\texttt{VAR V: } T}^{T}$ and $\boxed{\texttt{VAR V: } T}$, the control variable is removed from the state, then reinserted as if it had been newly declared, with the effect that upon exit from the FOR statement

V is undefined.

A FOR statement using DOWNTO is defined in a similar fashion.

17.2.3 Block Letters

As an extended example of the use of FOR statements, consider the problem of printing large block letters without explicitly locating each character that forms them. In an area that contains each block letter, say five columns wide by five lines high, number the positions as follows:

	Col 1	Col 2	Col 3	Col 4	Col 5
Line 1	1	2	3	4	5
Line 2	6	7	8	9	10
Line 3	11	12	13	14	15
Line 4	16	17	18	19	20
Line 5	21	22	23	24	25

In this area, a block letter may be represented as the set of positions in which a character is to be printed, the other positions to remain blank. For example, a block letter M might be represented by the set constant:

```
[1,5,6,7,9,10,11,13,15,16,20,21,25]
```

to indicate printing as follows:

```
M   M
MM MM
M M M
M   M
M   M
```

Thus, the type SET OF 1..25 can be used for the values that describe a block letter.

Two nested FOR statements (one to print each line and within it another to print each column) can be used to print the letters.

Design Part 1

```
PROCEDURE PrintLetter(Ch: CHAR);
  CONST
    Height = 5;
    Width = 5;
  TYPE
    PrintPos = SET OF 1..25;
  VAR
    BLetters: PrintPos;
  BEGIN {PrintLetter}
    {BLetters assigned set of print positions};
    {for each element of BLetters, print Ch}
  END;  {PrintLetter}
```

The value of Ch is used to select an assignment statement that assigns the appropriate set of print positions to BLetters.

Design Part 1.1

```
{BLetters assigned set of print positions}
CASE Ch OF
  'A': BLetters := [3,7,9,11..15,16,20,21,25];
  ...
  'M': BLetters := [1,5,6,7,9,10,11,13,15,16,20,21,25];
  ...
  'Z': BLetters := [1..5,9,13,17,21..25]
END
```

Two variables (HCount and WCount) are declared to keep track of which line and column are being printed.

Design Part 1.2

```
VAR
  HCount: 1..Height;
  WCount: 1..Width;
{for each element of BLetters, print Ch}
FOR HCount := 1 TO Height
DO
  BEGIN
    FOR WCount := 1 TO Width
    DO
      {if current position is in BLetters print Ch;
       otherwise print blank};
    WRITELN
  END
```

To determine whether to print the value of Ch or a blank, the values of the control variables (which both range from 1 to 5) must be mapped into the range 1 to 25 corresponding to the proper position for each row and column. Each increase in HCount must increase the value of the expression enough to skip the remaining print positions on a line. Multiplying the value of HCount by Width in the following expression accomplishes this.

```
(HCount-1) * Width + WCount
```

The next design part completes the procedure.

Design Part 1.2.1

```
{if current position is in BLetters print Ch;
 otherwise print blank}
IF (HCount-1)*Height+WCount IN BLetters
THEN
  WRITE(Ch)
ELSE
  WRITE(' ')
```

Assembling these design parts in a small driver program permits the design to be tested.

```
PROGRAM BlockLetter(OUTPUT);
  {Include PROCEDURE PrintLetter(Ch: CHAR);}
  BEGIN {BlockLetter}
    PrintLetter('A');
    WRITELN;
    PrintLetter('M')
  END.  {BlockLetter}
```

Execution

```
OUTPUT:   A
         A A
        AAAAA
        A   A
        A   A

        M   M
        MM MM
        M M M
        M   M
        M   M
```

17.2.4 Analysis of FOR Statements

Unlike the WHILE statement, the FOR statement is guaranteed to terminate. Thus no separate argument about loop termination is needed. To verify that a FOR statement computes a particular function, the FOR statement is expanded according to its definition and the function of the sequence of statements is worked out explicitly. As an example, consider the statement:

```
FOR I := 1 TO N
DO
  S := S + I
```

Its meaning is the following function (for `S`, omitting the control variable `I`):

```
(S := S + (N*(N+1)) DIV 2)
```

First the function of the FOR statement can be rewritten using its definition.

$$\boxed{\texttt{FOR I := 1 TO N DO S := S + I}} = \boxed{\texttt{I := 1}} \circ \boxed{\texttt{S := S + I}} \circ \boxed{\texttt{I := SUCC(I)}} \circ \boxed{\texttt{S := S + I}} \circ \cdots \boxed{\texttt{I := N}} \circ \boxed{\texttt{S := S + I}} \circ \boxed{\texttt{VAR I: INTEGER}}^{T} \circ \boxed{\texttt{VAR I: INTEGER}}$$

The compositions can be worked out using a trace table:

Function	I	S
I := 1	1	
S := S+I		S+1
I := SUCC(I)	SUCC(1)=2	
S := S+I		S+1+2
...		
I := N	N	
S := S+I		S+1+2+...+(N-1)+N

(The table stops just before `I` is removed from the state then reinserted.) Thus the result is:

```
(S := S+1+2+...+(N-1)+N)
```

The sum of the first `N` positive integers can be grouped from the beginning and end as follows:

```
1+N + 2+(N-1) + 3+(N-2) + ...
```

where there are $N \div 2$ pairs, since the terms are being removed two at a time from the original sum. Each pair sums to `N+1`, so the result is

```
(S := S + (N*(N+1)) DIV 2)
```

as required.

17.2.5 Exercises

17.2.1 The following declarations are in force for this question:

```
TYPE
  Drink = (Coffee, Tea, Milk);
VAR
  Ch: CHAR; Itg: 3..33;
  Enm: Drink;
  EFile: FILE OF Drink;
FUNCTION Ff(VAR Pr1: CHAR; Pr2: CHAR): BOOLEAN;
```

In each part below there is at least one mistake. Locate and explain each one.

a)
```
FOR Enm := PRED(Enm) DOWNTO SUCC(Enm)
DO
  WRITELN('HELLO')
```

b)
```
FOR Itg := 22 TO 33
DO
  Itg := SUCC(Itg)
```

17.2.2 In the following FOR statement what is the type of the variable `Qu` ?

```
FOR Qu := FALSE TO TRUE DO WRITELN('+')
```

How many lines are printed?

17.2.3 Explain why FOR statements are not allowed to have control variables of `SET` types.

17.2.4 Give a general method for stepping through the contents of a `SET` using a FOR statement. Does the type of the set members matter?

17.2.5 Explain why each legal FOR statement has to terminate (or abort), no matter what statements its body contains.

17.2.6 Modify `AverageScore` (Section 17.2.1) so that:

a) it handles 7 scores per line instead of 5.

b) it handles a 10-character name at the beginning of each line of scores, which is printed with the individual average for that line.

17.2.7 What is printed by the following fragment in which `Ending`, `Count`, and `RealCount` have been declared `INTEGER`?

```
Ending := 2;
RealCount := 0;
FOR Count := 0 TO Ending
DO
  BEGIN
    WRITELN(RealCount, Count, Ending);
    Ending := -2
  END
```

17.2.8 The box functions of both FOR and WHILE statements are defined as a k-fold composition of the box functions of statements that are repeated by these constructions, where k is the number of repetitions. A guess-and-verify method is required for calculating `WHILE ...`. Is a similar method needed for calculating `FOR ...`? Explain.

17.2.9 Consider the declarations:

```
TYPE
  Roman = (I, II, III, IV, V);
FUNCTION AddRoman(A, B: Roman): Roman;
  {Adds Roman numerals up to five}
  BEGIN {AddRoman}
    FOR A := I TO A
      DO
        BEGIN
          WRITELN('-');
          B := SUCC(B)
        END;
    AddRoman:= B
  END   {AddRoman}
```

a) If this function is used by

```
IF AddRoman(II, II) = IV
THEN
  WRITELN(' OK')
```

what will be added to OUTPUT?

b) Explain exactly the sense in which the function "adds Roman numerals up to five", by giving its precise meaning.

c) Suppose the parameters of the function were both variable parameters, and a Roman variable Tmp was declared. Discuss the code

```
Tmp := II;
IF AddRoman(Tmp, Tmp) = IV
THEN
  WRITELN(' OK')
```

17.2.10 Prove that the following program:

```
PROGRAM Sums(OUTPUT);
  CONST
    N = 10;
  VAR
    I,Odds,Evens: INTEGER;
  BEGIN {Sums}
    Odds := 0;
    Evens := 0;
    FOR I := 1 to N
    DO
      IF ODD(I)
      THEN
        Odds := Odds + I
      ELSE
        Evens := Evens + I;
    WRITELN(Odds,Evens)
  END.
```

produces the output:

```
          25          30
```

Using the method of Section 17.2.4.

17.3 The REPEAT Statement

Preview

REPEAT statements control iteration with a test at the end of the loop body, a test that terminates the iteration when TRUE.

The REPEAT statement provides for repetitive execution of a list of statements so long as a BOOLEAN expression controlling execution evaluates to FALSE. The form is:

```
REPEAT
  {Body statements}
UNTIL {controlling expression}
```

No BEGIN-END is required to bracket the statements of the loop body. Loop termination does not occur as soon as the controlling expression becomes TRUE, but when it evaluates to true at the end of the list of statements. Thus a REPEAT statement always has an iteration count of at least one. The fragment:

```
Exp := TRUE;
REPEAT
  WRITELN('This line is written to OUTPUT')
UNTIL Exp
```

appends a string to OUTPUT in spite of an expression controlling loop execution that is initially true.

17.3.1 REPEAT Statement Definition

The syntax rules for the REPEAT statement are shown below.

<repeat statement> ::= REPEAT <statement list> UNTIL <expression>

<statement list> ::= <statement list> ; <statement> | <statement>

A REPEAT statement:

REPEAT *S* UNTIL *B*

has exactly the same effect as a WHILE statement with the test reversed and the body performed initially:

BEGIN *S* END; WHILE NOT *B* DO BEGIN *S* END

The equivalence can be seen by examining cases. First, the REPEAT statement body *S* is executed exactly once if condition *B* is TRUE after *S* is executed. In the WHILE statement sequence, *S* is executed and the WHILE statement prevents further execution of *S* because NOT *B* is FALSE. Thus the two programs agree in this case. Second, suppose the REPEAT statement executed its body exactly twice. Then *B* is FALSE after the first execution of *S*, but something in *S* causes *B* to become TRUE when evaluated after a second execution of *S*, terminating the loop. In the WHILE sequence exactly the same behavior is observed. *S* is executed; NOT *B* is TRUE so the copy of *S* within the WHILE statement is executed, making *B* TRUE; then NOT *B* evaluates to FALSE, terminating the loop. The remaining cases in which the REPEAT statement is executed more than twice are similar: the copy of *S* preceding the WHILE statement takes the first execution, and the repeated *S* picks up the remainder.

The meaning of the REPEAT statement can be derived as was that for the WHILE statement. Consider

REPEAT S_1; S_2; ... S_n UNTIL *B*

where *B* is a BOOLEAN expression and each of the S_i, $1 \leq i \leq n$, are statements. Let *S* be an equivalent BEGIN statement:

BEGIN S_1; S_2; ... S_n END

and thus consider the REPEAT statement R:

REPEAT S UNTIL B

For any initial execution state s, R will either either have $k \geq 1$ iterations, or fail to terminate at all. If the statement fails to terminate, s is not a member of domain($[\![R]\!]$) and so of no concern. If R terminates, B must evaluate to FALSE after each of the first k-1 executions of S and to TRUE after its k^{th} execution.

The function for R can thus be defined as

$$[\![R]\!] = \{<s, t>: \exists\ k \text{ integer}, k \geq 1\ \forall\ i \text{ integer}, 1 \leq i < k\ [\![S]\!]^i \circ [\![B]\!](s) = \text{FALSE},$$
$$[\![S]\!]^k \circ [\![B]\!](s) = \text{TRUE},\ [\![S]\!]^k(s) = t\}.$$

A verification rule similar to that for a WHILE statement can be derived for a REPEAT statement. However, since each REPEAT statement has an equivalent statement sequence involving a WHILE statement, the former can be analyzed by analyzing the latter, as in Section 8.2.3.

17.3.2 Exercises

17.3.1 Consider the programs:

```
PROGRAM RepeatTest(INPUT, OUTPUT);
  VAR
    Color: (Red, Green, Blue);
  BEGIN {RepeatTest}
    Color := Red;
    REPEAT
      Color := SUCC(Color)
    UNTIL Color = Blue;
    WRITELN('Color is now blue')
  END.  {RepeatTest}
```

```
PROGRAM WhileTest(INPUT, OUTPUT);
  VAR
    Color: (Red, Green, Blue);
  BEGIN {WhileTest}
    Color := Red;
    WHILE Color <> Blue
    DO
      Color := SUCC(Color);
    WRITELN('Color is now blue')
  END.  {WhileTest}
```

a) Are they equivalent?

b) Suppose that the initial assignment is changed to

```
Color := Blue
```

in each program. Are they equivalent now?

17.3.2 By writing an equivalent WHILE program and analyzing it using the WHILE verification rule, show that the fragment R:

```
REPEAT
  Y := Y - 1;
  X := X + 2
UNTIL Y <= 0
```

computes the function:

```
f = (Y>0 → X,Y := X+2*Y,0) | (Y<=0 → X+2,Y-1)
```

17.3.3 Verify that the following program:

```
PROGRAM Divide(OUTPUT);
  CONST
    Z = 20;
    D = 3;
  VAR
    Q,R: INTEGER;
  BEGIN {Divide}
    Q := 0;
    R := Z;
    REPEAT
      Q := Q + 1;
      R := R - D
    UNTIL R<D;
    WRITELN(Q,R)
  END.  {Divide}
```

produces the output:

```
6          2
```

by writing an equivalent WHILE program and using the WHILE verification rule to analyze it.

17.3.4 Work out a REPEAT verification rule similar to the WHILE verification rule of Section 8.2.1.

17.4 Chapter Summary

A CASE statement is a convenient replacement for nested IF statements when an expression must be compared to several constant values, and a statement executed for each possibility. FOR statements control iteration whose limits can be established before the iteration begins. They provide a control variable whose value shows the position through the complete iteration. The REPEAT statement is a variant of the WHILE statement in which the test is reversed and placed at the end of each iteration. Each of these statements is sometimes more natural to use than the equivalent IF or WHILE statement.

CHAPTER 18

ABSTRACT DATA TYPES

Chapter Preview

The program modules described in Chapter 13 are important because they divide the complexity of a program. In this chapter it is shown that they are also useful in dividing the calculation of program meaning. The meaning obtained for a module may be used in an application, and the calculation outside the module need not be repeated if hidden changes are made at the concrete level. A method is given for verifying that a module correctly implements an abstract meaning.

When it was helpful to count in CF Pascal, lacking the natural data type for counting (INTEGER), Chapter 13 created a module containing a collection of CHAR variables and the procedures to use them for counting. All the cumbersome aspects of using characters to count were hidden within this counter module, so that module users could ignore them. When the INTEGER type was introduced in Chapter 14, the counter module was no longer required. But from the user's point of view there is little difference: the counting operations using INTEGER are essentially the same as those using the module. In this sense INTEGER is not a *necessary* feature of Pascal. (The limitations of size imposed by the counter module were seen in Chapter 14 to be inherent in the INTEGER type as well; they are limitations inherent in mechanical computation.)

The situation with a counter and INTEGER is not typical for Pascal or any other language. The complicated data objects needed for difficult problems usually do not appear magically in a later chapter. Thus programmers must construct the objects they need, but by hiding them in modules they can be made so easy to use that they might as well be built in. Pascal features that were not available in Chapter 13 make module construction far easier and more powerful:

1. A TYPE declaration can be used for objects within the module, to hide their detailed construction behind the type identifier.
2. A RECORD can be used in the TYPE to bind several objects into a new aggregate representing an abstract object.

The module makes the TYPE construction details its secret, exporting the name only. Users of the module can declare objects of this type, but are not supposed to know how they are constructed. The rule of module construction is that for a TYPE secret, the TYPE *identifier* is available outside the module, but the details of the TYPE *definition* are *not* available.

As an example, recall the counter example of Chapter 13, with some changes to take advantage of TYPE declarations. The changes will allow users to have as many distinct counters as needed, and to start a counter off at values other than zero. Users think of Counter objects as integer values in the range 0-999, but within the module they are characters, the module's secret. By declaring:

```
TYPE
  Counter = RECORD
              Hundreds, Tens, Ones: CHAR
            END
```

users can create the needed characters without ever mentioning more than the TYPE identifier. For example,

```
VAR
  C1, C2: Counter
```

would create two counters. To preserve the secret, the user must use only the names C1 and C2, not (for example) C2.Tens, etc. To get the characters to the module, its procedures must be given a Counter parameter:

```
PROCEDURE StartAt(VAR Ctr: Counter; D100, D10, D1: CHAR);
PROCEDURE Bump(VAR Ctr: Counter);
PROCEDURE Value(Ctr: Counter; VAR D100, D10, D1: CHAR);
```

where the `Start` of Chapter 13 has been replaced by `StartAt` which allows the counter to be initialized from the hundreds, tens, and units digit values in the corresponding `CHAR` parameters. Then a user declaring

```
VAR
  C1, C2: Counter
```

might call

```
StartAt(C2, '1', '0', '3');
Bump(C2);
 ...
Bump(C2);
Value(C2, Hd, Td, Od);
WRITELN('The value is ', Hd, Td, Od)
```

to use `C2` beginning at 103, and could use `C1` for some different counting task.

The illustrated use of a `Counter` is the general case for modules. The module contains two special procedures that get things started (here, `StartAt`) and report results (here, `Value`), and one or more procedures (here just one, `Bump`) that modify things in between. The pattern is to begin with regular Pascal objects (here, characters) that are supplied to the starting procedure, then perform intermediate manipulations (without any detailed knowledge of the objects being manipulated), and finally to obtain the result, again as regular Pascal objects. Of course, the initial values might have been obtained with a READ statement or in some other way, and the final values might have been used in some other way than with a WRITE statement. But to make use of them outside the module requires that they be converted from/to standard Pascal data objects.

The use of a `TYPE` declaration also protects the module interface against changes in the data structures used within the module. If the components of the `TYPE` were shown explicitly, the declaration for `Bump` would have been:

```
PROCEDURE Bump(VAR Hundreds, Tens, Ones: CHAR)
```

but if the module were to switch to using `INTEGER` it would be:

```
PROCEDURE Bump(VAR Count: INTEGER)
```

such a change would require the user to alter not only the calls, but declarations of quantities to be passed as actual parameters. On the other hand, if the `TYPE` declaration alone is changed:

```
TYPE
  Counter = INTEGER
```

then changes are confined to the module; the user program declares and uses `Counter` objects without caring that they have changed form.

18.1 Concrete and Abstract Worlds

Preview

The implementors of a module and its users have very different views of the module meaning, which are related but distinct.

It is important to keep the two worlds, the one within a module, and the one outside for users of the module, distinct. We call the module interior the *concrete* world, and the user one outside the *abstract* world. It is difficult to keep these apart, since they are intimately connected by a certain idea that the user wants the module to realize. But if modules are to be useful in hiding the details of an abstraction, the separation must be observed.

The user's view of a module is called an *abstract data type*. It consists of the set of *abstract objects* that the user imagines to be available, and this set is named by the TYPE identifier from the module. The user imagines that the abstract objects have a standard description consisting of other Pascal objects. Finally, the user has in mind a collection of *abstract operations* that manipulate abstract objects. Two of these operations are special: one must map the standard objects that describe an abstract object to the latter; the other maps in the reverse direction, from an abstract object to its corresponding standard objects.

In the counting example, the abstract objects are integers in the range 0-999, and the TYPE identifier is Counter. The standard description of a counter comprises three characters for its digits. The three operations in the example are:

StartAt, the special operation carrying a standard description of three characters to a Counter;

Value, the other special operation, carrying a Counter back to three characters; and

Bump, which maps a Counter to a Counter.

This view provides a user with everything needed to create objects of the abstract data type (using the special mapping from known objects), manipulate these abstract objects (using the operations), and then pass back from the abstract results to known forms (using the other special mapping).

From the viewpoint of the module itself things are quite different. We say that the module *implements* an abstract data type *T* when it contains a TYPE definition for the identifier of *T*, and procedures for each of the abstract mappings.

In the counting example the TYPE definition is

```
TYPE
  Counter: RECORD
             Hundreds, Tens, Ones: CHAR
           END
```

and the procedures are StartAt, Value, and Bump.

In the concrete world of the module, there are only ordinary Pascal data objects and procedures that manipulate them. When a parameter of the abstract TYPE is passed into this world, it is really made up of component objects according to the TYPE definition. The procedures manipulate these components using regular Pascal operations. In the abstract world, the user imagines that abstract objects exist, those that have been declared to have the abstract TYPE, and that the abstract operations map them one to another. In the counting example, if the user declares

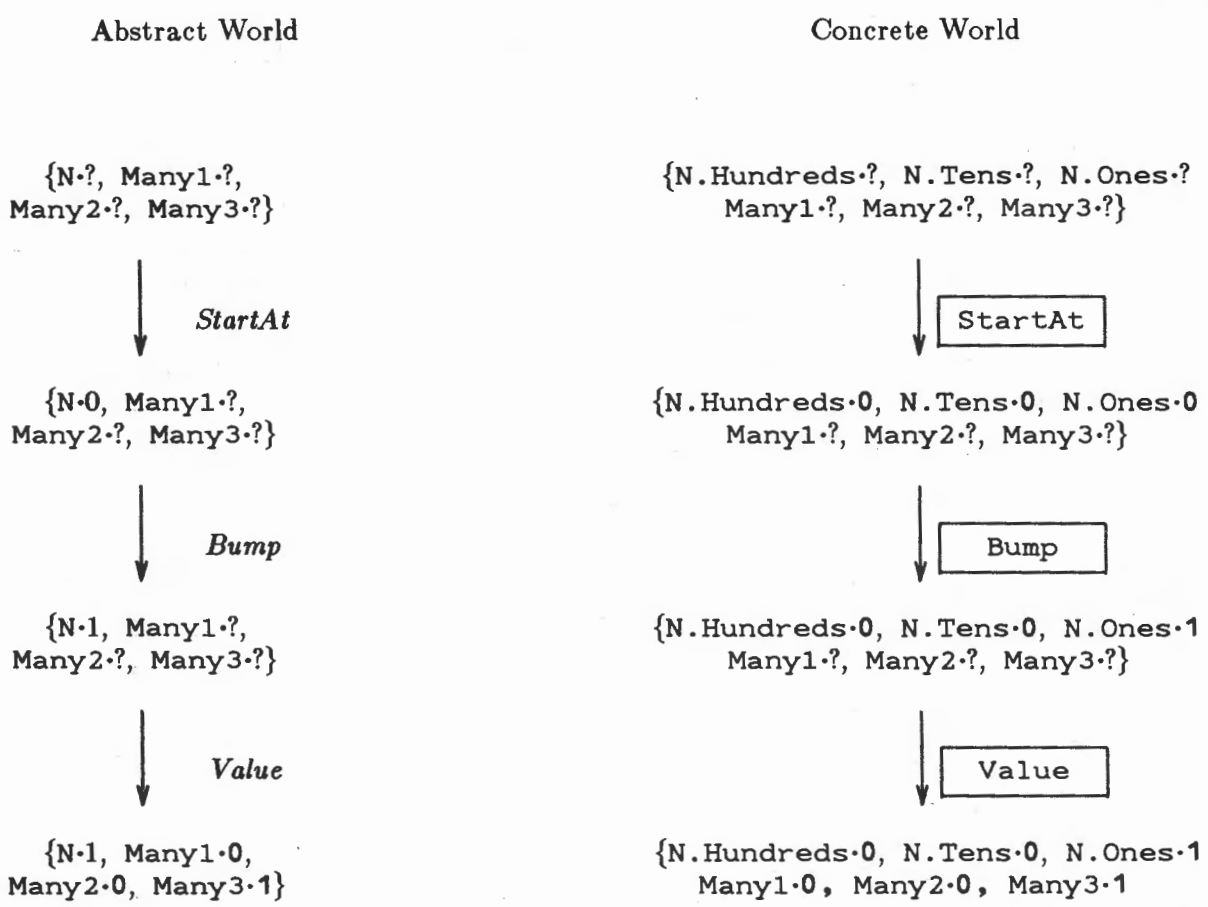
```
VAR
  N: Counter;
  Many1, Many2, Many3: CHAR
```

then the abstract world has data states containing four identifiers and their values, while the concrete world lacks identifier N and its imagined integer value, having instead the variables N.Hundreds, N.Tens, and N.Ones, with their character values. When the user performs a sequence of operations like

```
StartAt(N, '0', '0', '0');
Bump(N);
Value(N, Many1, Many2, Many3)
```

the abstract picture is that the Counter object 0 is created from character constants, then increased to 1, and transformed into the character triple (0, 0, 1). In the concrete world, three CHAR variables (N.Hundreds, N.Tens, and N.Ones) are initially set to 0, the last of these is altered to 1, and the final values transferred to other CHAR variables (Many1, Many2, Many3).

The situation in the two worlds can be diagramed as follows:



If the procedures are doing the right things, the actions in the concrete world properly reflect those in the abstract world step by step. The correspondence is the subject of Section 18.3.

18.2 Rational Numbers

Preview

Rational numbers (fractions) are a data abstraction not available in most computer languages. They can be defined in a module, and used to solve a problem involving a fractional series.

The *harmonic series* has the partial sum

$$H(n) = 1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n.$$

The terms are fractions, so H(n) is a fraction. To calculate H(n) exactly requires operating in the abstract world on objects that are rational numbers; these will be implemented in the concrete world as pairs of INTEGER values.

The first design part for calculating H(n) shows as comments parts that cannot be executed by the D Pascal machine. This design part reads a single integer from the input and prints the sum of that many terms from the series.

Design Part 1

```
PROGRAM HarmonicSeries (INPUT, OUTPUT);
{There is an integer n in INPUT,
  (n>0 --> Sum := 1 + 1/2 + 1/3 + ... + 1/n) |
  (n<=0 --> Sum := 0) }
  VAR
    Terms: INTEGER;
    Ctr: INTEGER;
    {Sum: Rational}
    DSum, NSum: INTEGER;
  BEGIN {HarmonicSeries}
    READ(Terms);
    Ctr := 0;
    {Sum := 0/1};
    {Get numerator and denominator of Sum
     in NSum and DSum};
    WRITELN('Term   0, Sum ', NSum, ' /', DSum);
    WHILE Ctr < Terms
    DO {Sum := 1 + 1/2 + ... + 1/Ctr}
      BEGIN
        Ctr := Ctr + 1;
        {Sum := Sum + 1/Ctr};
        {Get numerator and denominator of Sum
         in NSum and DSum};
        WRITELN('Term  ', Ctr:2,
                ', Sum ', NSum, ' /', DSum)
      END
  END.  {HarmonicSeries}
```

18.2.1 Representing Rationals

A rational number can be conveniently implemented as a pair of integers, which are grouped in a record so they can be passed as a single parameter.

```
TYPE
  Rational = RECORD
               Num, Den: INTEGER
             END;
```

Declaring `Rational` as a type identifier permits the declarations of Design Part 1 to be completed.

Design Part 1.1

```
{Sum: Rational}
Sum: Rational;
```

18.2.2 Operations on Rationals

The operations needed for the harmonic series problem are implemented in procedures `MakeRat`, `AddRat`, and `BackRat`. `MakeRat` converts two integers into a rational number; `AddRat` adds two rationals and produces a rational value; and `BackRat` converts a rational back into a pair of integers. Since `MakeRat` and `AddRat` return rational values, they would be best implemented as functions. Then Design part 1.2 could be refined into a single Pascal statement:

Design Part 1.2

```
{Sum := Sum + 1/Ctr}
Sum := AddRat(Sum,MakeRat(1,Ctr)) {Not allowed in Pascal}
```

As the comment indicates, Pascal does not permit functions to return any type but an ordinal, so the `Rational RECORD` does not qualify. Thus, `MakeRat` and `AddRat` must be procedures which return their values through variable parameters. A temporary variable `R` to hold the result from a procedure simulating a function must be added to the declarations and the following sequence of instructions must be used:

Design Part 1.2

```
{Sum := Sum + 1/Ctr}
BEGIN
  MakeRat(R,1,Ctr);   {R := 1/Ctr}
  AddRat(Sum,Sum,R)   {Sum := Sum + R}
END
```

Having made these decisions, the remaining design parts are easy to complete.

Design Part 1.3

```
{Sum := 0/1}
MakeRat(Sum,0,1)
```

Values returned by `BackRat` can be written out directly.

Design Part 1.4

```
{Get numerator and denominator of Sum
 in NSum and DSum}
BackRat(Sum, NSum, DSum)
```

The module for rational numbers is an example of the general definitions given in Section 18.1. The abstract data type consists of

1) the `TYPE` identifier `Rational`;
2) the set of rational numbers (fractions), with the standard description a pair of `INTEGER` values for numerator and denominator;
3) the operations of constructing a `Rational` from two `INTEGER` values, adding two `Rational` values to get a third, and converting back to two `INTEGER` values.

This is the abstract point of view.

The implementation consists of the `RECORD` with fields `Num` and `Den` for `Rational` and the procedures `MakeRat`, `AddRat`, and `BackRat` now to be developed. This is the concrete point of view.

The concrete world is related to the abstract in that concrete states containing `X.Num` and `X.Den` (but not `X` itself) correspond to abstract states containing `X` (but not `X.Num` and `X.Den`), for all identifiers `X` declared `Rational` in the abstract world. The rational value associated with such an abstract identifier `X` is the one whose numerator and denominator are the values respectively associated with `X.Num` and `X.Den` in the concrete world.

In writing an implementation of the `Rational` data type, it is essential to keep in mind what the user is expecting, and correspondence between the two worlds. In this simple case the declaration of `TYPE Rational` is reminder enough of the representation mapping, but in more complex cases it may help to first write the special conversion routines for the implementation, which usually explore all the ins and outs of the connection. The expectations of the user can be captured by writing comments about the procedure part functions in the abstract state. These comments are labeled "abs:" in the code. Of course, the concrete procedures themselves are concrete-state mappings, and have part functions in that world as usual; these are described by comments labeled "con:".

Although HarmonicSeries needs only nonnegative rational numbers, we will implement a module to handle negative ones as well.

BackRat can simply return its parameter's Num and Den components for numerator and denominator as long as both these values are positive. However, when the Num or Den values are negative there is the possibility of multiple forms for the same rational number. Most people would consider (-1)/4 and 1/(-4) to be the same fraction, but if BackRat simply returns the Num and Den values both could appear. A *canonical form* (or standard notation) for rational numbers will be adopted to avoid this duplication. Positive rational numbers will be represented with positive numerators and denominators and negative rational numbers will be represented with negative numerators and positive denominators. If the operations that produce rational values (MakeRat and AddRat) adhere to this convention, BackRat can just return the existing values.

```
PROCEDURE BackRat(VAR Rat: Rational;
                  VAR NumRat, DenRat: INTEGER);
  {abs: there are n,d such that Rat = n/d -->
        NumRat, DenRat := n, d }
  {con: NumRat, DenRat := Rat.Num, Rat.Den }
  BEGIN {BackRat}
    NumRat := Rat.Num;
    DenRat := Rat.Den
  END;  {BackRat}
```

MakeRat creates the internal form for a Rational from its INTEGER components. The concrete comment is more complex than the abstract comment because it must describe the canonical form used in the implementation.

```
PROCEDURE MakeRat(VAR Result: Rational;
                  Num, Den: INTEGER);
  {abs: (Den<>0 --> Result := Num/Den) |
        (Den=0  --> Result := 0/1)     }
  {con:
   (Den<0  -->
      Result.Num,Result.Den := -Num,-Den) |
   (Den=0  -->
      Result.Num,Result.Den := 0,1)       |
   (Den>0  -->
      Result.Num,Result.Den := Num,Den)   }
  BEGIN {MakeRat}
    IF Den < 0
    THEN
      BEGIN
        Result.Num := -Num;
        Result.Den := -Den
      END
    ELSE {Den >= 0}
      IF Den > 0
      THEN
        BEGIN
          Result.Num := Num;
          Result.Den := Den
        END
      ELSE {Den = 0}
        BEGIN
          WRITELN;
          WRITELN('*** zero Den for MakeRat ***');
          Result.Num := 0;
```

```
        Result.Den := 1
      END
  END;  {MakeRat}
```

Notice the action taken for the case of a zero denominator: MakeRat issues an error message, but creates a(n incorrect) concrete value anyway. This is described in both the abstract and concrete comments, and is appropriate when software is being debugged. Experimentation can continue after the error occurs. However, when everything is working correctly, it might be better to cause MakeRat to abort after the error message instead of creating a value. That way a program *cannot* continue after such an error occurs.

The following formula can be used to add two Rational values, contained in Op1 and Op2:

$$\frac{\texttt{Op1.Num} * \texttt{Op2.Den} + \texttt{Op1.Den} * \texttt{Op2.Num}}{\texttt{Op1.Den} * \texttt{Op2.Den}}$$

AddRat must be written very carefully because it can be used with the same actual parameter for two of its formal parameters. That happens in HarmonicSeries, where Sum is both one of the values to be added, and the place to store the result. The following ill-considered code:

```
PROCEDURE AddRat(VAR Result, Op1, Op2: Rational);
  BEGIN {AddRat}
    {Ill-considered because of parameter aliasing}
    Result.Den := Op1.Den * Op2.Den;
    Result.Num := Op1.Num * Op2.Den +
                  Op1.Den * Op2.Num
  END;  {AddRat}
```

goes wrong when Result and (say) Op1 are given the same identifier as actual parameter, because then the assignment to Result.Den is really changing Op1, whose value is used in the second assignment. In that case the value for Result is actually:

$$\frac{\texttt{Op1.Num} * \texttt{Op2.Den} + \texttt{Op1.Den} * \texttt{Op2.Den} * \texttt{Op2.Num}}{\texttt{Op1.Den} * \texttt{Op2.Den}}$$

However, since the calculation for Result.Den uses no .Num values, the statements can be reversed to avoid this difficulty:

```
PROCEDURE AddRat(VAR Result, Op1, Op2: Rational);
  {abs: Result := Op1 + Op2 }
  {con: Result.Den,Result.Num :=
           Op1.Den * Op2.Den,
           Op1.Num * Op2.Den + Op1.Den * Op2.Num }
  BEGIN {AddRat}
    Result.Num := Op1.Num * Op2.Den +
                  Op1.Den * Op2.Num;
    Result.Den := Op1.Den * Op2.Den
  END;  {AddRat}
```

This implementation of AddRat makes no explicit provision for producing results in canonical form, but an inductive argument proves that this happens automatically. Assuming that Rational input parameters are in canonical form (i.e., have positive denominators), it can be shown that Rational outputs are also in canonical form. Only MakeRat and AddRat produce rational outputs so the results of these procedures must be shown to be in canonical form. Since MakeRat has no rational inputs, it is the base case. MakeRat contains explicit statements to guarantee that the resulting Den is always positive. The inductive hypothesis lets us assume that the input parameters for AddRat, Op1 and Op2, have positive denominators. Since Op1.Den and Op2.Den are both positive, Op1.Den*Op2.Den is also positive. Thus the Rational result of AddRat always has a positive denominator.

18.2.3 Harmonic Series

The assembled solution for HarmonicSeries is shown below.

```
PROGRAM HarmonicSeries (INPUT, OUTPUT);
{There is an integer n in INPUT,
  (n > 0 --> Sum := 1 + 1/2 + 1/3 + ... + 1/n) |
  (n <= 0 --> Sum := 0) }
  TYPE
    Rational = RECORD
                 Num, Den: INTEGER
               END;
  VAR
    Terms: INTEGER;
    Ctr: INTEGER;
    Sum, R: Rational;
    DSum, NSum: INTEGER;
  {Include PROCEDURE BackRat
     (VAR Rat: Rational, VAR NumRat, DenRat: INTEGER);
     abs: there are n,d such that Rat = n/d -->
          NumRat, DenRat := n, d }
  {Include PROCEDURE MakeRat
     (VAR Result: Rational; Num, Den: INTEGER);
     abs: (Den<>0 --> Result := Num/Den) |
          (Den=0  --> Result := 0/1)     }
  {Include PROCEDURE AddRat
     (VAR Result, Op1, Op2: Rational);
     abs: Result := Op1 + Op2 }
  BEGIN {HarmonicSeries}
    READ(Terms);
    Ctr := 0;
    {Sum := 0/1}
    MakeRat(Sum, 0, 1);
    {Get numerator and denominator of Sum
     in NSum and DSum}
    BackRat(Sum, NSum, DSum);
    WRITELN('Term   0, Sum ', NSum, ' /', DSum);
    WHILE Ctr < Terms
    DO {Sum := 1 + 1/2 + ... + 1/Ctr}
      BEGIN
        Ctr := Ctr + 1;
        {Sum := Sum + 1/Ctr}
        BEGIN
          MakeRat(R,1,Ctr);  {R := 1/Ctr}
          AddRat(Sum,Sum,R)  {Sum := Sum + R}
        END;
        {Get numerator and denominator of Sum
         in NSum and DSum}
        BackRat(Sum, NSum, DSum);
        WRITELN('Term  ', Ctr:2,
                ', Sum ', NSum, ' /', DSum)
      END
  END.  {HarmonicSeries}
```

Execution

```
INPUT :13
OUTPUT:Term   0, Sum           0 /          1
       Term   1, Sum           1 /          1
       Term   2, Sum           3 /          2
       Term   3, Sum          11 /          6
       Term   4, Sum          50 /         24
       Term   5, Sum         274 /        120
       Term   6, Sum        1764 /        720
       Term   7, Sum       13068 /       5040
       Term   8, Sum      109584 /      40320
       Term   9, Sum     1026576 /     362880
       Term  10, Sum    10628640 /    3628800
       Term  11, Sum   120543840 /   39916800
       Term  12, Sum  1486442880 /  479001600
       Term  13, Sum -1672077440 / 1932053504
```

The 13-term sum above is obviously incorrect since overflow occurred in the `AddRat` formulas for this case (and perhaps for earlier ones, less obviously). (This Pascal machine exceeds *MAXINT* silently—it just begins delivering the wrong result.) If it were necessary to sum more terms, the canonical form of rational numbers could be changed so their numerators and denominators had no common divisors. For example, by removing the common divisor 2, term 4 above can be represented as 25/12.

18.2.4 An Alternate Representation for Rationals

In order to remove common divisors from rational numbers, we first compute the greatest common divisor of two integers with a function `Gcd`.

```
FUNCTION Gcd(I1, I2: INTEGER): INTEGER;
  { I1>0 AND I2>0 --> Gcd := GCD(I1,I2) |
    I1<=0 AND I2>0 --> Gcd := I2 |
    I2<=0 --> Gcd := I1 }
  VAR
    Rem: INTEGER;
  BEGIN {Gcd};
    WHILE I2 > 0
    DO
      BEGIN
        Rem := I1 MOD I2;
        I1 := I2;
        I2 := Rem
      END;
    Gcd := I1
  END;  {Gcd}
```

Positive values of `I2` are the only ones allowed to enter the WHILE statement because `MOD` is undefined if its second operand is not positive. If `I1` is not positive, the result of `I1 MOD I2` will not be positive either. Then the WHILE statement will execute exactly once since the result of the `MOD` operation is assigned to `I2` each time the body is executed, and `I1` will be assigned the value of `I2`.

To check the correctness of `Gcd`, determine the function of the body:

`WHILE I2 > 0 ...` ∘ `Gcd := I1`

and compare it to that given in the comment:

```
(I1>0 AND I2>0 → Gcd := GCD(I1,I2)) |
(I1<=0 AND I2>0 → Gcd := I2) |
(I2<=0 → Gcd := I1)
```

where *GCD* is the mathematical function for the greatest common divisor. The meaning F of the WHILE statement is obtained from the comment and the program itself:

```
F = (I1>0 AND I2>0 → I1,I2 := GCD(I1,I2),0) |
      (I1<=0 AND I2>0 → I1,I2 := I2, I1 MOD I2) |
      (I2<=0 → )
```

(The temporary variable `Rem` has been omitted, since it is only used to preserve a value in the calculation of `I1` and `I2`.) The WHILE statement verification rule requires establishing the three conditions:

1. domain(F) = domain(`WHILE I2 > 0 ...`)
2. `(I2<=0 →` F`)` = `(I2<=0 → )`
3. $F =$ `IF I2>0 THEN` $D \circ F$

where D is the BEGIN statement within the WHILE statement.

For the first condition, it is clear that the conditions on `I1` and `I2` in F cover all possibilities, so F is defined for all states in which these variables are defined. The WHILE statement terminates for all values of `I1` and `I2`. There is no iteration for values of `I2` that are not positive. If `I2` is positive, consider the possibilities for `I1`. If `I1` ≤ 0, `(I1 MOD I2)` ≤ 0 and since this value is assigned to `I2` the WHILE statement terminates after a single execution of its body. If `I1` > 0, `I2` is always decreasing in the loop because it is set to `I1 MOD I2` each time through the loop and the value of `I1 MOD I2` is strictly less than the value of `I2`. Thus the domain of the WHILE statement function and the domain of F are the same.

For the second condition, the guard `I2<=0` selects the identity case, so the result is immediate.

Finally, in the third condition, the function corresponding to the IF statement in the right side is:

```
(I2>0 → I1,I2 := I2, I1 MOD I2) | (I2<=0 → )
```

To work out the right side:

```
((I2>0 → I1,I2 := I2, I1 MOD I2) | (I2<=0 → )) ∘
((I1>0 AND I2>0 → I1,I2 := GCD(I1,I2),0) |
(I1<=0 AND I2>0 → I1,I2 := I2, I1 MOD I2) |
(I2<=0 → ))
```

six cases must be considered.

Part	Condition	I1	I2
`IF I2>0 ...`	`I2>0`	`I2`	`I1 MOD I2`
F	`I2>0 AND I1 MOD I2>0`	*GCD*`(I2,I1 MOD I2)`	0

Thus the function is:

```
I2>0 AND I1 MOD I2>0 → I1,I2 := GCD(I2,I1 MOD I2),0
```

Part	Condition	I1	I2
`IF I2>0 ...`	`I2>0`	`I2`	`I1 MOD I2`
F	`I2<=0 AND I1 MOD I2>0`	`I1 MOD I2`	`I2 MOD (I1 MOD I2)`

The composite condition includes `I2>0 AND I2<=0`,which cannot be satisfied. Thus this case contributes no pairs to the resulting function.

Part	Condition	I1	I2
IF I2>0 ...	I2>0	I2	I1 MOD I2
F	I1 MOD I2<=0		

Thus the function is:

```
I2>0 AND I1 MOD I2<=0 → I1,I2 := I2, I1 MOD I2
```

Part	Condition	I1	I2
IF I2>0 ...	I2<=0		
F	I1>0 AND I2>0	*GCD*(I1,I2)	0

Again the composite condition requires the impossible of I2, so there is no contribution to the function.

Part	Condition	I1	I2
IF I2>0 ...	I2<=0		
F	I1<=0 AND I2>0	I2	I1 MOD I2

This case also contributes no pairs to the function.

Part	Condition	I1	I2
IF I2>0 ...	I2<=0		
F	I2<=0		

Thus the function is:

```
(I2<=0 → )
```

The union of these six functions yields the following:

```
(I2>0 AND I1 MOD I2>0 → I1,I2 := GCD(I2,I1 MOD I2),0) |
(I2>0 AND I1 MOD I2<=0 → I1,I2 := I2, I1 MOD I2) |
(I2<=0 → )
```

Some properties of greatest common divisors are needed to continue.

Property 1: *GCD*(X,Y) = *GCD*(Y,X)
Property 2: *GCD*(X,Y) = *GCD*(X MOD Y, Y)
Property 3: If X MOD Y = 0, then Y = *GCD*(X,Y).

Applying Property 1 to the first part function gives:

```
(I2>0 AND I1 MOD I2>0 → I1,I2 := GCD(I1 MOD I2,I2),0) |
(I2>0 AND I1 MOD I2<=0 → I1,I2 := I2, I1 MOD I2) |
(I2<=0 → )
```

Applying Property 2 next results in:

```
(I2>0 AND I1 MOD I2>0 → I1,I2 := GCD(I1,I2),0) |
(I2>0 AND I1 MOD I2<=0 → I1,I2 := I2, I1 MOD I2) |
(I2<=0 → )
```

Splitting the second function into two cases corresponding to I1 MOD I2=0 and I1 MOD I2<0 yields:

```
(I2>0 AND I1 MOD I2>0 → I1,I2 := GCD(I1,I2),0) |
(I2>0 AND I1 MOD I2=0 → I1,I2 := I2,0) |
(I2>0 AND I1 MOD I2<0 → I1,I2 := I2, I1 MOD I2) |
(I2<=0 → )
```

By Property 3:

```
(I2>0 AND I1 MOD I2>0 → I1,I2 := GCD(I1,I2),0) |
(I2>0 AND I1 MOD I2=0 → I1,I2 := GCD(I1,I2),0) |
(I2>0 AND I1 MOD I2<0 → I1,I2 := I2, I1 MOD I2) |
(I2<=0 → )
```

Now the first two cases can be combined:

```
(I2>0 AND I1 MOD I2>=0 → I1,I2 := GCD(I1,I2),0) |
(I2>0 AND I1 MOD I2<0 → I1,I2 := I2, I1 MOD I2) |
(I2<=0 → )
```

This function is identical to F except for the guards. However we observe:

```
(I2>0 AND I1 MOD I2>=0) ≡ (I2>0 AND I1>0)
(I2>0 AND I1 MOD I2<0) ≡ (I2>0 AND I1<0)
```

so the two functions are really identical.

With F established as the function of the WHILE statement, the entire procedure has the meaning $F \circ$ `Gcd := I1`:

```
(I1>0 AND I2>0 → Gcd,I1,I2 := GCD(I1,I2),GCD(I1,I2),0) |
(I1<=0 AND I2>0 → Gcd,I1,I2 := I2,I2,I1 MOD I2) |
(I2<=0 → Gcd := I1)
```

This function is the same as that of the comment on Gcd, except that it includes changes to the value parameters I1 and I2. These disappear outside of the procedure. Thus Gcd does calculate the greatest common divisor in the case of positive input parameters.

18.2.5 Re-implemented Operations for Rationals

Only the procedures MakeRat and AddRat need to use Gcd. Notice that neither procedure calls Gcd unless both actual parameters to the function are strictly positive, thus confining the procedure to the case for which it has been shown to compute *GCD*. (The code to arrange for positive values occurs three times, so it probably should have been packaged with the Gcd function rather than repeated on each call.)

```
PROCEDURE MakeRat(VAR Result: Rational;
                  Num, Den: INTEGER);
  {abs: (Den<>0 --> Result := Num/Den) |
        (Den=0  --> Result := 0/1)      }
  {con: Let x = gcd(Num,Den) then
       (Den<0 -->
          Result.Num,Result.Den := -Num/x,-Den/x) |
       (Den=0 -->
          Result.Num,Result.Den := 0,1) |
       (Den>0 -->
          Result.Num,Result.Den := Num/x,Den/x) }
  VAR
    Cf: INTEGER;
  BEGIN {MakeRat}
    {find the gcd of Num and Den}
    IF Num * Den <> 0
    THEN
      Cf := Gcd(ABS(Num),ABS(Den))
    ELSE
      Cf := 1;
    IF Den < 0
    THEN
      BEGIN
        Result.Num := -Num DIV Cf;
        Result.Den := -Den DIV Cf
```

```
    END
  ELSE
    IF Den > 0
    THEN
      BEGIN
        Result.Num := Num DIV Cf;
        Result.Den := Den DIV Cf
      END
    ELSE
      BEGIN
        WRITELN;
        WRITELN('*** zero Den for MakeRat ***');
        Result.Num := 0;
        Result.Den := 1
      END
END;  {MakeRat}
```

Gcd is called a second time in AddRat after Sum is computed because the addition of two reduced rational terms does not necessarily result in a reduced rational result. For example, consider the case in which the sixth term is added to the sum of the first five terms: $\frac{137}{60} + \frac{1}{6}$. The denominators have a common factor of 6 which is removed and used to multiply their product:

$$\frac{137 * (6 \text{ DIV } 6) + 1 * (60 \text{ DIV } 6)}{10 * 1 * 6} = \frac{147}{60}.$$

A common factor of 3 can be removed from the result to obtain the sixth term in the series: $\frac{49}{20}$.

```
PROCEDURE AddRat(VAR Result, Op1, Op2: Rational);
  {abs: Result := Op1 + Op2 }
  {con: Let x = gcd(Op1.Den*Op2.Den,
                    Op1.Num*Op2.Den+Op2.Num*Op1.Den),
        Result.Den,Result.Num :=
          (Op1.Den * Op2.Den) / x,
          (Op1.Num*Op2.Den + Op1.Den*Op2.Num) / x }
  VAR
    Cf: INTEGER;
    Den1, Den2: INTEGER;
  BEGIN {AddRat}
    {find the gcd of Op1.Den and Op2.Den}
    IF Op1.Den*Op2.Den <> 0
    THEN
      Cf := Gcd(ABS(Op1.Den),ABS(Op2.Den))
    ELSE
      Cf := 1;
    Den1 := Op1.Den DIV Cf;
    Den2 := Op2.Den DIV Cf;
    Result.Num := Op1.Num * Den2 + Den1 * Op2.Num;
    Result.Den := Den1 * Den2 * Cf;
    {find the gcd of Result.Num and Result.Den}
    IF Result.Num*Result.Den <> 0
    THEN
      Cf := Gcd(ABS(Result.Num),ABS(Result.Den))
    ELSE
      Cf := 1;
    Result.Num := Result.Num DIV Cf;
    Result.Den := Result.Den DIV Cf
```

```
    END;  {AddRat}
```

The user need not be concerned with any of the Gcd transformations—all the user notices is that more terms in the series can be computed before overflow occurs. Since rational numbers were implemented as a module, the code for the BEGIN statement in HarmonicSeries does not change.

Execution

```
INPUT :25
OUTPUT:Term   0, Sum          0 /         1
       Term   1, Sum          1 /         1
       Term   2, Sum          3 /         2
       Term   3, Sum         11 /         6
       Term   4, Sum         25 /        12
       Term   5, Sum        137 /        60
       Term   6, Sum         49 /        20
       Term   7, Sum        363 /       140
       Term   8, Sum        761 /       280
       Term   9, Sum       7129 /      2520
       Term  10, Sum       7381 /      2520
       Term  11, Sum      83711 /     27720
       Term  12, Sum      86021 /     27720
       Term  13, Sum    1145993 /    360360
       Term  14, Sum    1171733 /    360360
       Term  15, Sum    1195757 /    360360
       Term  16, Sum    2436559 /    720720
       Term  17, Sum   42142223 /  12252240
       Term  18, Sum   14274301 /   4084080
       Term  19, Sum  275295799 /  77597520
       Term  20, Sum   55835135 /  15519504
       Term  21, Sum   18858053 /   5173168
       Term  22, Sum   19093197 /   5173168
       Term  23, Sum  444316699 / 118982864
       Term  24, Sum 1347822955 / 356948592
       Term  25, Sum  307215901 /-333780208
```

18.2.6 Exercises

18.2.1 Change the abstract comment for MakeRat to include the changes made to OUTPUT in the case Den=0.

18.2.2 Alter the action of MakeRat in the case of zero denominator to assign *no* values in this impossible case; that is, to abort. Change both abstract and concrete comments to reflect what you have done.

18.2.3 Aliasing of parameters had to be considered in AddRat. Is there a similar problem with parameters Rat and NumRat in the operation BackRat? Explain.

18.2.4 Suppose that for some Pascal machine the value of *MAXINT* is defined by CONST MxI.

a) Explain how to alter the Rational module so that erroneous results like those shown in the executions of HarmonicSeries never occur. [Hint: use a subrange type.]

b) Will any changes be required in HarmonicSeries itself? Explain.

c) Change the abstract comments to reflect this change.

18.2.5 Consider altering the Rational module so that rational values may be externally described as an integer part plus a fraction. For example, instead of passing the pair (5, 3) to the module representing five thirds, a triple (1, 2, 3) could be passed representing one and two thirds.

a) Design an appropriate input-conversion procedure for this format, making no assumptions about the fractional part. Do not change the internal form for `Rational` values, and use `MakeRat` in the implementation if you can.

b) Discuss the question of whether or not to alter the internal form for `Rational` values to have an integer, numerator, and denominator part. Describe a situation in which this form would be the most efficient, and another in which it would not be efficient. Consider also the question of making the internal form unique—when would that be wise, and when not wise?

c) Design an appropriate output-conversion procedure for the new format, assuming that the internal form for `Rational` values is not changed. Return the largest possible integer part (that is, with a fractional part less than 1).

d) Show the changes in `HarmonicSeries` that would be required to print the result in the form of an integer plus fraction, using your designs from a) and c).

18.3 The Calculus of Abstract Data Types

Preview

The user of an abstract data type can employ its operations correctly if the "abs:" comments can be trusted. It is shown how to verify that these do indeed reflect the actual actions taken by an implementation. Furthermore, the abstract comments can be used to calculate the meaning of user programs. Thus these comments are all that users need to know about an abstract data type.

To illustrate the calculus of an abstract data type, we will use a particularly simple abstraction. The program below determines if the input contains an even or an odd number of characters (counting line markers as single characters). It uses an abstract `Switch` object `Sw` to keep track of the characters read. `Sw` has abstract values *Off* and *On*. The abstract operations for a `Switch` are *Init*, *Flip*, and *Test*. *Init*(x, y) initializes the value of `Switch` x from the `CHAR` value `Y` (*On*) or `N` (*Off*) in y. *Flip*(x) changes x from *Off* to *On* and vice versa. *Test*(x, y) queries the value of `Switch` x and sets y to the corresponding `CHAR` value.

```
PROGRAM OddOrEven (INPUT,OUTPUT);
  { (INPUT = <,w,R> and ODD(length(w)) -->
       OUTPUT := <'Input has odd length',,W> ) |
    (INPUT = <,w,R> and EVEN(Length(w)) -->
       OUTPUT := <'Input has even length',,W> ) }
  TYPE
    Switch = RECORD
               S: CHAR
             END;
  VAR
    Ch, SwitchOn: CHAR;
    Sw: Switch;
```

```
PROCEDURE Init(VAR Sw: Switch; Ival: CHAR);
  {abs: (Ival='Y' --> Sw := On) |
        (Ival<>'Y' --> Sw := Off }
  {con: (Ival='Y' --> Sw.S := 'U') |
        (Ival<>'Y' --> Sw.S := 'D') }
  BEGIN {Init}
    IF Ival = 'Y'
    THEN
      Sw.S := 'U'
    ELSE
      Sw.S := 'D'
  END;  {Init}
PROCEDURE Flip(VAR Sw: Switch);
  {abs: (Sw=On --> Sw := Off) |
        (Sw=Off --> Sw := On) }
  {con: (Sw.S='U' --> Sw.S := 'D') |
        (Sw.S<>'U' --> Sw.S := 'U')}
  BEGIN {Flip}
    IF Sw.S = 'U'
    THEN
      Sw.S := 'D'
    ELSE
      Sw.S := 'U'
  END;  {Flip}
PROCEDURE Test(VAR Sw: Switch; VAR Result: CHAR);
  {abs: (Sw=On --> Result := 'Y') |
        (Sw=Off --> Result := 'N')}
  {con: (Sw.S='U' --> Result := 'Y') |
        (Sw.S<>'U' --> Result := 'N') }
  BEGIN {Test}
    IF Sw.S = 'U'
    THEN
      Result := 'Y'
    ELSE
      Result := 'N'
  END;  {Test}
BEGIN {OddOrEven}
  Init(Sw,'N');
  WHILE NOT EOF
  DO
    BEGIN
      READ(Ch);
      Flip(Sw)
    END;
  Test(Sw,SwitchOn);
  IF SwitchOn = 'Y'
  THEN
    WRITELN('Input has odd length')
  ELSE
    WRITELN('Input has even length')
END.  {OddOrEven}
```

By the rules of module isolation, no program part except the operations of the abstract type may access or alter the variables that represent an abstract object. Thus, the variable `Sw.S` is not used

in the BEGIN statement of OddOrEven. Two concurrent assignment comments describe the affect of each procedure. The assignment comment labeled "abs:" deals with the abstract objects known to the user (i.e., Sw), and the assignment labeled "con:" uses concrete objects (i.e., Sw.S) that are hidden from the user.

The definition of Section 18.1 applied to this module gives the following:

1) The abstract data type consists of the TYPE identifier Switch, and the set of abstract objects {*On*, *Off*}. The standard description of the value *On* is the character Y, and of *Off* is the character N. (An enumerated type (Off, On) might have been a better choice for the standard description, but we used characters to emphasize the distinction between abstract values and their standard descriptions.) For abstract operations, the *Init* operation is the special one that creates abstract values from their standard descriptions; the *Test* operation is the inverse operation; *Flip* maps the abstract values into themselves by reversing the value of its parameter.
2) The implementation consists of the TYPE definition for Switch, namely: RECORD S: CHAR END, and the procedures Init, Flip, and Test, which would be functions returning Switch values but for the Pascal restriction to return only ordinal types.
3) The representation mapping onto the abstract value for identifier Sw is defined for two values of the concrete identifier Sw.S: U (carried to *On*), and D (carried to *Off*). (Again, an enumerated type might have been a better choice, but we wanted to clearly distinguish the concrete values both from the abstract ones, and from the standard description.)

This example is too small to offer much abstraction, but OddOrEven is isolated from changes in the implementation. If an Australian had written the implementation, the values D and U would probably be reversed, since Australian light switches are down when on, up when off. Such changes in the values for the .S component of the implementation do not alter the BEGIN statement of OddOrEven at all.

18.3.1 Verifying an Implementation

The "abs:" comments are added to modules so that users, those in the abstract world, need not examine the code (or even the "con:" comments that document it). But abstract comments are very different than others placed with code: there is no code to back them up! There are no "abstract procedures," and all the ideas about the abstract data type are intuitive, in the minds of human users, not in the implementation. The question that then arises is: how can a user know that the abstract comment is trustworthy? A user with this question is in the hands of the person who implemented the module. The implementor knew what the user was expecting, and in fact wrote the abstract comment. If the implementation has been done properly, the abstract comment can be believed, and used in proofs at the abstract level. The formal definitions of Section 18.1 allow us to *prove* that this is the case.

An abstract data type can only be used as follows: starting with standard descriptions of abstract objects, the user calls the input conversion function, which returns the corresponding abstract objects for further manipulation; finally, abstract values are given to the output conversion function and standard descriptions returned to the user. That is, the user views the data abstraction as a composite transformation from an input standard description to an output standard description. Since only the standard-description objects exist outside the module, the user *cannot* manipulate any abstract objects without first converting; and, the results of abstract operations *cannot* be observed without converting back. This limited pattern of use is the basis for the definition of module correctness.

Consider a typical such application of the Switch module in which a Switch X is initialized from a standard description, then flipped, and finally returned as a standard description. That is, the abstract operations *Init*, *Flip*, *Test* are applied in sequence:

Init(X, 'N') ∘ *Flip*(X) ∘ *Test*(X, R).

X is started with a value whose standard description is N. *Init* then gives X the abstract value *Off*, *Flip* transforms this to *On*, and *Test* returns the standard description Y in variable R. From the

abstract point of view, the module has transformed **N** to **Y**, which the user thinks of as *Off* to *On*.

This same sequence of operations can be viewed from the concrete side, within the module. The sequence of operations is

$$\boxed{\texttt{Init(X, 'N')}} \circ \boxed{\texttt{Flip(X)}} \circ \boxed{\texttt{Test(X, R)}}$$

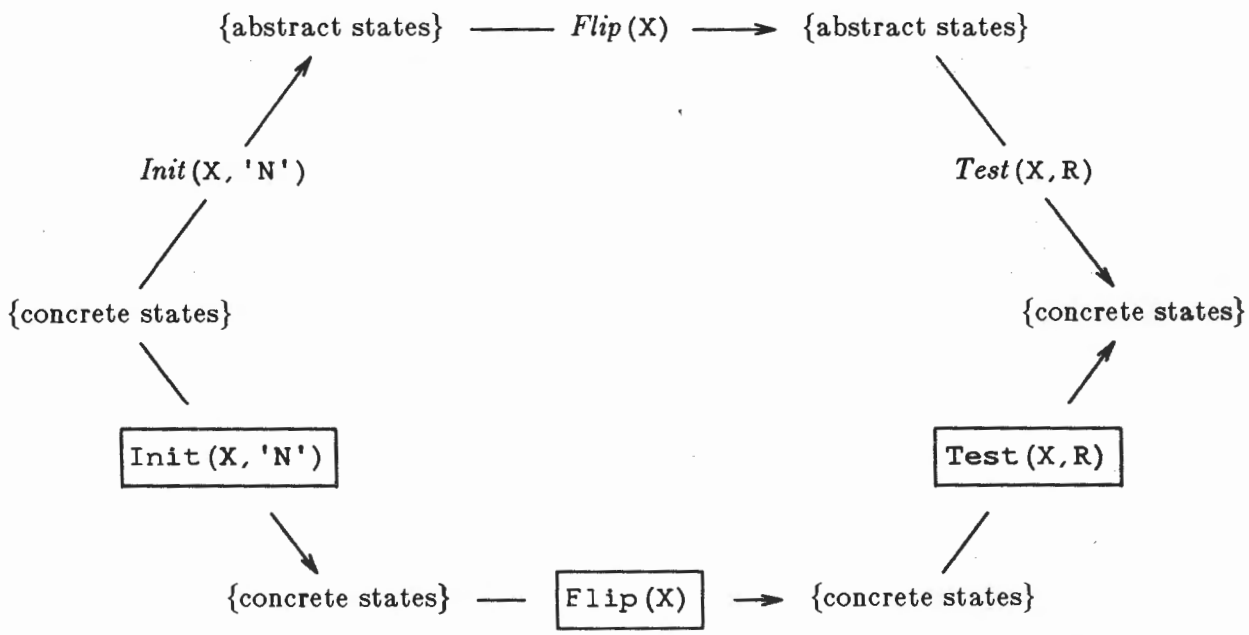
which results in an assignment to `X.S` of **U**, changing this to **D**, and finally returning **Y** in variable R. Thus from the concrete point of view, the module has transformed **N** to **Y**, but done so through the hidden values **D** to **U** of `X.S`.

The sequence of concrete functions $\boxed{\texttt{Init(X, 'N')}}$, $\boxed{\texttt{Flip(X)}}$, $\boxed{\texttt{Test(X, R)}}$ *correctly implements* the sequence of abstract operations *Init*(X, 'N'), *Flip*(X), *Test*(X, R) if and only if, for any (concrete) state T,

$$(\mathit{Init}(\texttt{X, 'N'}) \circ \mathit{Flip}(\texttt{X}) \circ \mathit{Test}(\texttt{X, R}))(T)$$
$$\subseteq (\boxed{\texttt{Init(X, 'N')}} \circ \boxed{\texttt{Flip(X)}} \circ \boxed{\texttt{Test(X, R)}})(T).$$

The definition does not insist that these compositions be exactly the same, only that when the abstract functions are defined, the concrete ones mimic them. It is not unusual for the concrete functions to be defined when the abstract are not. For example, instead of aborting in some error case, a module may issue an error message; this was done in the `Rational` module for zero denominators.

The sequence of operations in each of the two worlds can be shown in a single *implementation diagram* beginning and ending with concrete states:



This implementation diagram *commutes*: starting at the left with any standard description, the standard description obtained at the right by following the upper (abstract) path is the same as the one obtained at the right by following the lower (concrete) path. Within the diagram the two worlds differ, but they begin and end in the same place. The statement that the diagram commutes is exactly the same as the statement that the sequence of concrete operations correctly implement the abstract sequence.

Commuting implementation diagrams can be used to reason about the relationship between the concrete and abstract worlds. No matter what initial standard description is chosen, and no matter what intermediate operations are used, the path through the abstract world should lead to the same final standard description as the path through the concrete world. For the `Switch` module this means considering all values of the parameter `Ival` passed to `Init`, and all sequences of operations beginning with `Init`, continuing with any number of `Flip` applications, and ending

with `Test`. For any such sequence the text of these procedures defines the resulting value for the parameter `Result` in `Test`. The concrete comments summarize what should happen. There is no "abstract code", but the abstract comments can be used to calculate `Result` from `Ival`. The implementation is correct if and only if these two sequences of operations agree.

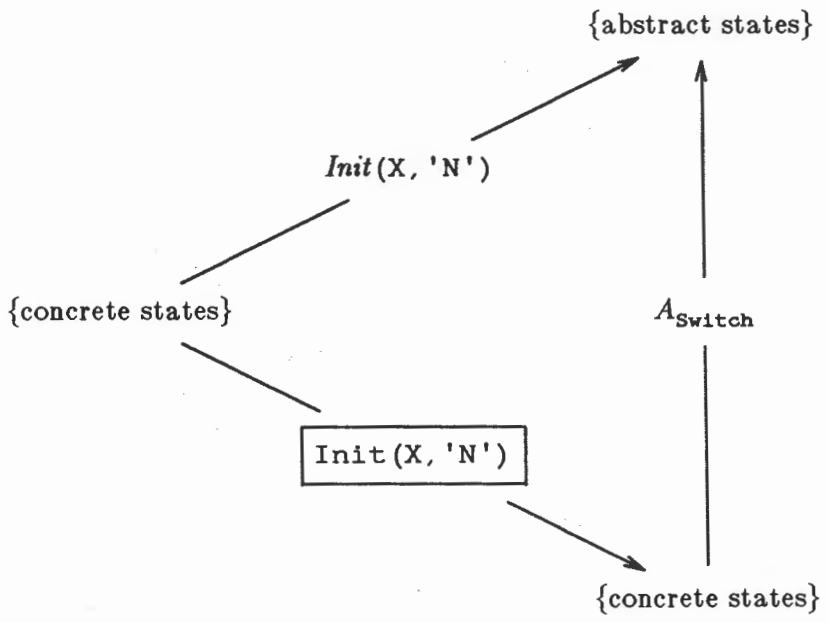
It is a simple matter to check the particular implementation diagram above, which represents one initial value for `Ival` and the simplest sequence of operations consisting of just one `Flip`. Tracking the assignment statements in the concrete and abstract worlds, it can be easily seen that the diagram does commute. However, to handle *all* diagrams in a complex module would be a formidable task. Fortunately, the proofs can be decomposed by considering a correspondence between the two worlds within each implementation diagram.

An implementation diagram must necessarily commute if the abstract and concrete worlds "stay in step." That is, they begin together with a common (concrete) state. If each function in turn maps to a state where there is still agreement, when they come back together in a common final state, that agreement will still exist. To make a connection between the worlds within the implementation diagram requires a new mapping called the *representation mapping*. This mapping is in the mind of the person doing the implementation, and reflects the intuitive correspondence between the two kinds of objects. It maps from the concrete to the abstract rather than the other way around, because while it is essential that every abstract object have at least one corresponding concrete object, it can happen that some concrete objects have no abstract counterparts, or that more than one concrete object corresponds to the same abstract one. If the representation mapping were to go from abstract to concrete, under these circumstances it would not be a function. For objects that lie in both worlds, that is, concrete objects that are not involved in the module's special TYPE declaration, the representation mapping is identity.

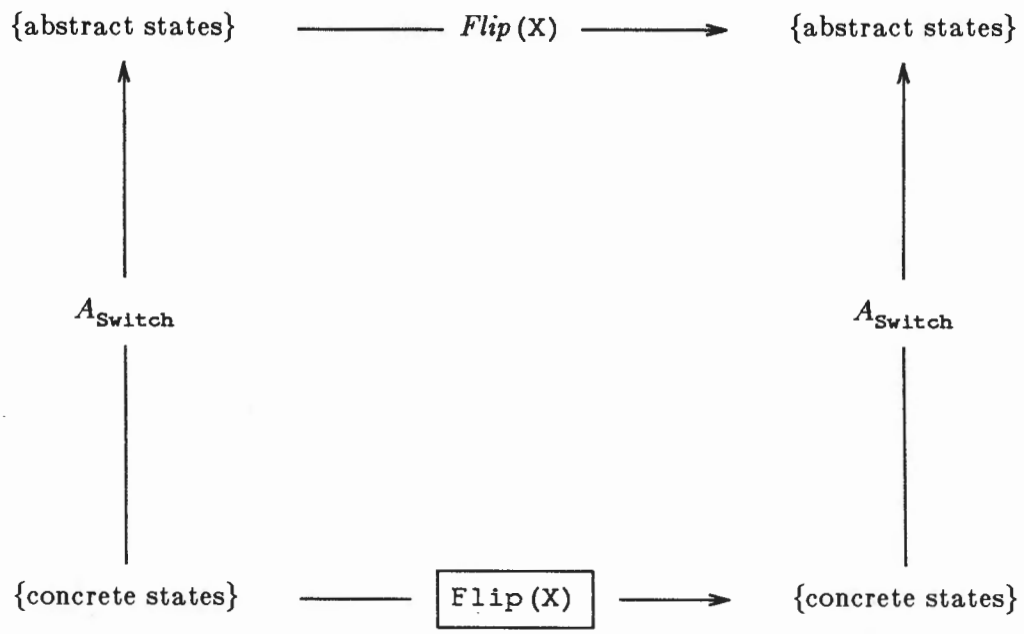
In the case of the `Switch` module, the representation mapping that the programmer had in mind is $A_{\texttt{Switch}}$, which carries any concrete identifier `X.S` and its value to the abstract identifier `X` and its value. Value `U` corresponds to *On*, and `D` to *Off*. That is,

$$A_{\texttt{Switch}} = \{<s, t>: t = s \text{ except that } \texttt{X} \text{ in } t \text{ replaces } \texttt{X.S} \text{ in } s,$$
$$\text{with } t(\texttt{X}) = \mathit{Off} \text{ if } s(\texttt{X.S}) = \texttt{D},\ t(\texttt{X}) = \mathit{On} \text{ if } s(\texttt{X.S}) = \texttt{U}\}.$$

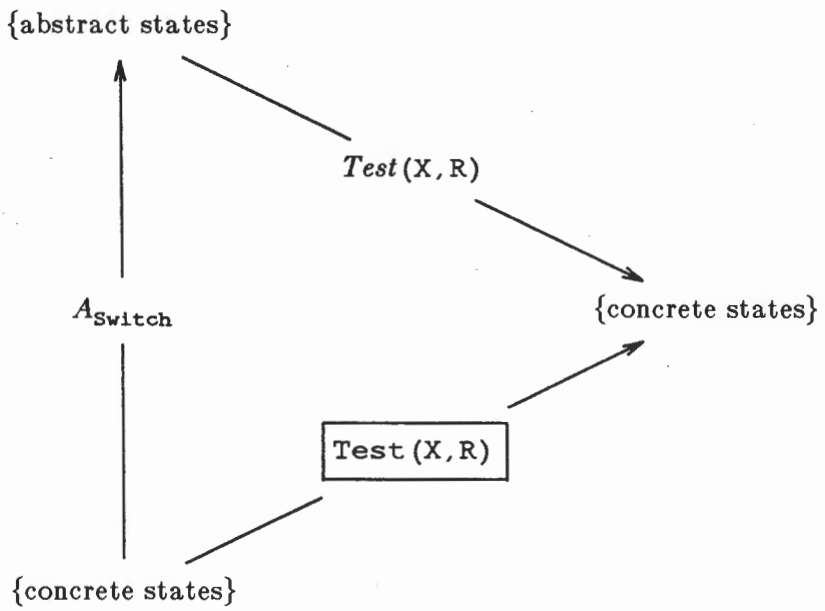
An implementation diagram can be decomposed using a representation mapping. For example, for `Switch`, the diagram



concerns only the input-conversion mapping, while



deals with the *Flip* operation alone, and



is for the output-conversion mapping. If each of these diagrams commutes starting at the (lower) left and ending at the (upper) right, then the composite implementation diagram must also commute.

All possible implementation diagrams can be handled by being slightly more general in the proofs of the individual-operation diagrams. If instead of using the particular parameters `X` and `'N'` for `Init`, we use its formal parameters `Sw` and `Ival`, then proving that the diagram commutes proves that `Init` is a correct implementation for *all* possible parameter values. If the diagrams for `Flip` and `Test` are similarly verified, then *any* implementation diagram beginning with `Init`, continuing with any number of applications of `Flip`, and ending with `Test`, must commute. That is, checking just three separate diagrams proves that the module is correctly implemented.

This discussion leads to the following method for verifying the correctness of a module:

1) Find a representation mapping A that is identity on concrete objects not in the module's TYPE, that maps the implementation RECORD identifiers to a single abstract identifier, with intuitively correct value correspondence.
2) Check that the diagram for each separate operation of the module commutes using A. This can be done by verifying that the concrete comments correctly describes the implemented functions, and then comparing with the function defined by the abstract comment.

The representation mapping is not part of the module implementation, but it is the most important piece of information the person doing the implementation possesses. Correctness turns on the existence of this mapping, making the individual diagrams commute.

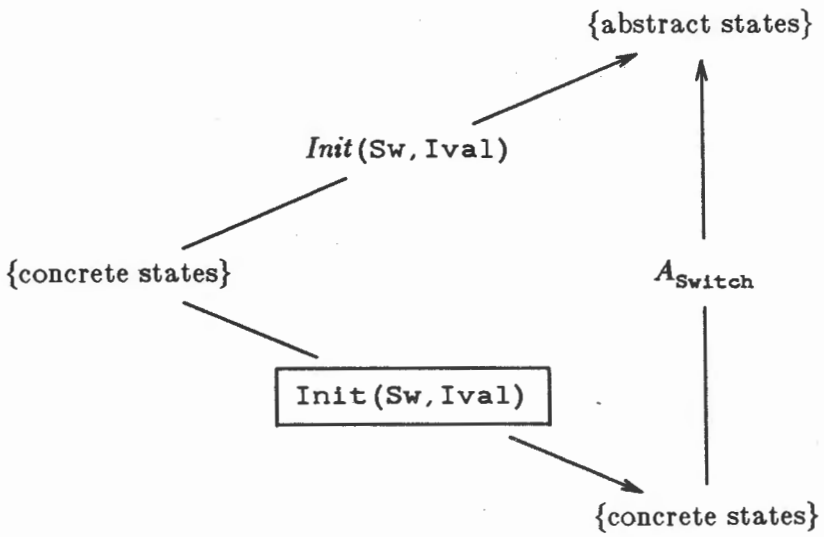
18.3.2 Sample Verification

As an example, the abstract data type `Switch` used in `OddOrEven` will be verified. A representation mapping was worked out above. For the identifier `Sw` that is used as parameter in all three procedures it is:

$$A_{\texttt{Switch}} = \{<s, t>: t = s \text{ except that } \texttt{Sw} \text{ in } t \text{ replaces } \texttt{Sw.S} \text{ in } s,$$
$$\text{with } t(\texttt{Sw}) = \mathit{Off} \text{ if } s(\texttt{Sw.S}) = \texttt{D},\ t(\texttt{Sw}) = \mathit{On} \text{ if } s(\texttt{Sw.S}) = \texttt{U}\},$$

which is identity on all parts of a state not involving the `Switch` `Sw`, as required. It remains to show that each operation has a commuting diagram using $A_{\texttt{Switch}}$. In the proofs it will be assumed that the concrete comment is correct, but of course this must be checked using the program calculus in the usual way within the module.

Consider `Init`, whose diagram is:



The statement that the diagram commutes is:

$$\mathit{Init}(\texttt{Sw, Ival}) \subseteq \boxed{\texttt{Init(Sw, Ival)}} \circ A_{\texttt{Switch}}.$$

The abstract and concrete comments give these functions. From the abstract comment,

```
Init (Sw, Ival) =
      (Ival='Y' → Sw := On) | (Ival<>'Y' → Sw := Off)
```

On the other hand, `Init(Sw, Ival)` is not defined by the concrete comment, but by the program calculus for the procedure code. We assume that it has been so calculated, and that it agrees with the concrete comment:

$\boxed{\texttt{Init(Sw, Ival)}} =$

(Ival='Y' → Sw.S := 'U') | (Ival<>'Y' → Sw.S := 'D')

The right side of the commutativity equation is:

$\boxed{\texttt{Init(Sw, Ival)}} \circ A_{\texttt{Switch}} =$

((Ival='Y' → Sw.S := 'U') | (Ival<>'Y' → Sw.S := 'D') ∘ ((Sw.S='U' → Sw := On) | (Sw.S='D' → Sw := Off))

The following execution tables help to compute this composition.

part	condition	Sw	Sw.S	Ival
Init(Sw,Ival)	Ival='Y'		U	
$A_{\texttt{Switch}}$	Sw.S='U'	*On*		

part	condition	Sw	Sw.S	Ival
Init(Sw,Ival)	Ival='Y'		U	
$A_{\texttt{Switch}}$	Sw.S='D'	*Off*		

part	condition	Sw	Sw.S	Ival
Init(Sw,Ival)	Ival<>'Y'		D	
$A_{\texttt{Switch}}$	Sw.S='U'	*On*		

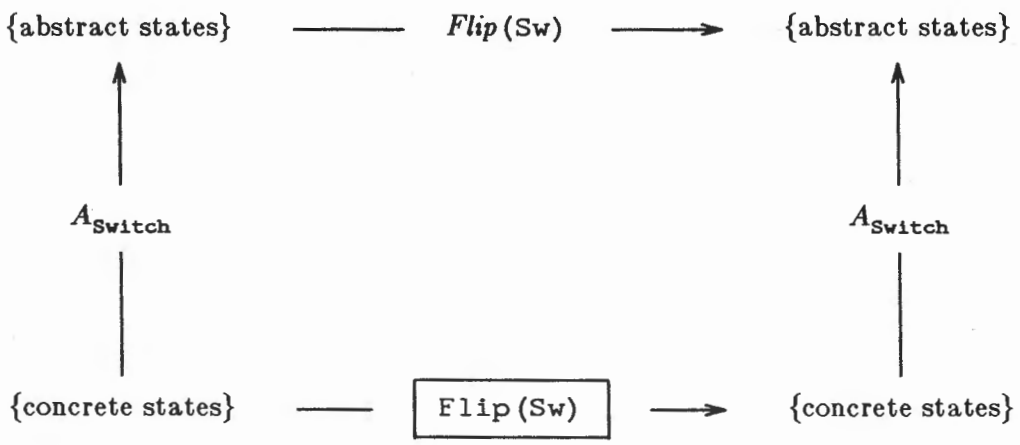
part	condition	Sw	Sw.S	Ival
Init(Sw,Ival)	Ival<>'Y'		D	
$A_{\texttt{Switch}}$	Sw.S='D'	*Off*		

The second part of the composite condition is TRUE (U = U and D = D) in the first and third tables, and FALSE in the second and fourth tables, so the result is:

(Ival='Y' → Sw := *On*) | (Ival<>'Y' → Sw := *Off*)

This is the same as *Init*(Sw, Ival), and hence the diagram for Init commutes.

The diagram for Flip is:



and the statement that it commutes is:

$$A_{Switch} \circ Flip(Sw) \subseteq \boxed{Flip(Sw)} \circ A_{Switch}.$$

Inserting the definition and abstract and concrete comments, this is:

```
((Sw.S='U' → Sw := On) | (Sw.S='D' → Sw := Off)) ∘
   ((Sw=On → Sw := Off) | (Sw=Off → Sw := On))
⊆ ((Sw.S='U' → Sw.S := 'D') | (Sw.S<>'U' → Sw.S := 'U')) ∘
   ((Sw.S='U' → Sw := On) | (Sw.S='D' → Sw := Off))
```

The trace tables for these compositions are similar to those for `Init`; only two cases survive in each case. The left side is:

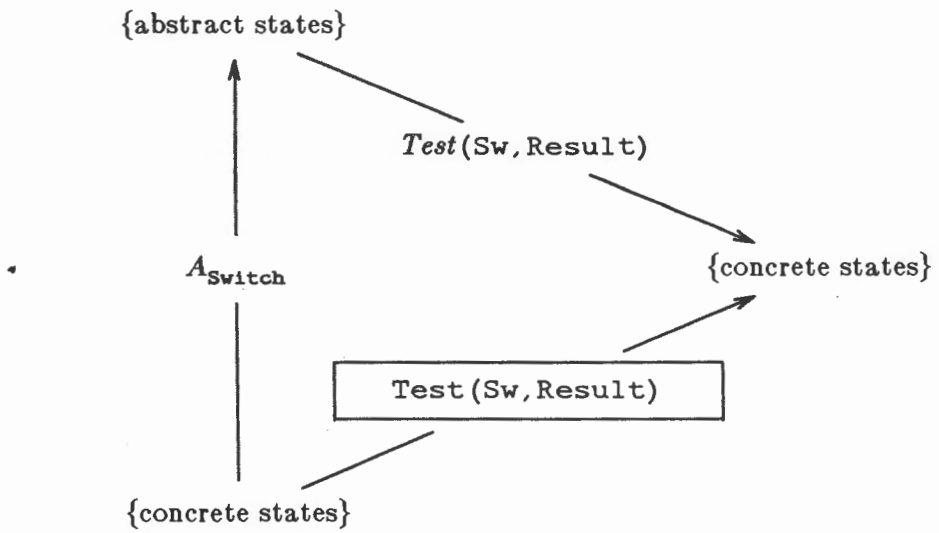
```
(Sw.S='U' → Sw := Off) | (Sw.S='D' → Sw := On)
```

and the right side is:

```
(Sw.S='U' → Sw := Off) | (Sw.S<>'U' → Sw := On)
```

The former is a subset of the latter as required, so the `Flip` diagram commutes.

The diagram for `Test` is:



The required equation is

$A_{\texttt{Switch}} \circ \mathit{Test}(\texttt{Sw, Result}) \subseteq$ `Test(Sw, Result)`,

or with the comments inserted:

```
((Sw.S='U' → Sw := On) | (Sw.S='D' → Sw := Off)) ∘
   ((Sw=On → Result := 'Y') | (Sw=Off → Result := 'N'))
⊆ (Sw.S='U' → Result := 'Y') | (Sw.S<>'U' → Result := 'N')
```

The left side is:

```
(Sw.S='U' → Result := 'Y') | (Sw.S='D' → Result := 'N')
```

which is identical to `Test(Sw, Result)`, except for having a smaller domain. Thus the `Test` diagram commutes.

Verification of the `Switch` module is complete: a representation function was found that makes all of the operation diagrams commute.

18.3.3 Verifying a Module Use

Verifying an implementation establishes the correctness of its abstract comments as a description of the abstract functions. Thus *Init*, *Flip*, and *Test* and the values *On* and *Off* can be employed in calculations about the program `OddOrEven` using the `Switch` module. Once the implementation is verified, none of its Pascal code, not even its concrete comments, need be examined again.

Ignoring line breaks, suppose the input to `OddOrEven` is a string w. The comment indicates that the function computed should be

$\{<w,$ †**Input has even length**†$>$: *length*(w) is even$\}$ ∪
$\{<w,$ †**Input has odd length**†$>$: *length*(w) is odd$\}$.

The calculation of the program function amounts to working out the following:

`BEGIN ... END` ∘ `VAR Ch, SwitchOn: CHAR; Sw: Switch`$^{\mathrm{T}}$ ∘ `.`

({`INPUT`·<††,w,**R**>, `OUTPUT`·(††,††,**W**), `Ch`·?, `SwitchOn`·?, `Sw`·?}).

Within the BEGIN statement,

`Init(Sw, 'N')` $= \{(r, s)$: $s = r$ except `Sw`$(s) = \mathit{Off}\}$.

The function computed by the WHILE statement is:

$\{<s, t>$: where $t = s$ except that `INPUT`(t)=<w,††,**R**>,
`Ch`(t)=□, `Sw`(t)=*Off* if *length*(w) is even and `Sw`(s)=*Off*$\}$ ∪

$\{<s, t>$: where $t = s$ except that `INPUT`(t)=<w,††,**R**>,
`Ch`(t)=□, `Sw`(t)=*Off* if *length*(w) is odd and `Sw`(s)=*On*$\}$ ∪

$\{<s, t>$: where $t = s$ except that `INPUT`(t)=<w,††,**R**>,
`Ch`(t)=□, `Sw`(t)=*On* if *length*(w) is odd and `Sw`(s)=*Off*$\}$ ∪

$\{<s, t>$: where $t = s$ except that `INPUT`(t)=<w,††,**R**>,
`Ch`(t)=□, `Sw`(t)=*On* if *length*(w) is even and `Sw`(s)=*On*$\}$.

(It must be shown that this is indeed `WHILE NOT EOF DO...` using the WHILE verification rule.)

Because `Init(Sw, 'N')` produces a state in which `Sw` has the value *Off*, only the first and third parts of the WHILE statement meaning appear in the composition of the two functions:

$\{<r, t>$: where $t = r$ except that `INPUT`(t)=<w,††,**R**>,
`Ch`(t)=□, `Sw`(t)=*On* if *length*(w) is odd$\}$ ∪

$\{<r, t>$: where $t = r$ except that `INPUT`(t)=<w,††,**R**>,
`Ch`(t)=□, `Sw`(t)=*Off* if *length*(w) is even$\}$.

Because there are no identifier conflicts and no aliasing, `Test(Sw,SwitchOn)` can be calculated by substituting `SwitchOn` for `Result` in the abstract comment for `Test`:

`Test(Sw,SwitchOn)` =

(Sw=*On* → SwitchOn := 'Y') | (Sw=*Off* → SwitchOn := 'N')

Combining these three functions:

`Init(Sw,'N')` ∘ `WHILE NOT EOF DO ...` ∘ `Test(Sw,SwitchOn)`

= {<*r*, *s*>: where *s* = *r* except that `INPUT`(*s*)=<*w*,††,**R**>,

`Ch`(*s*)=□, `Sw`(*s*)=*On*, `SwitchOn`(*s*)=**Y** if *length*(*w*) is odd} ∪

{<*r*, *s*>: where *s* = *r* except that `INPUT`(*s*)=<*w*,††,**R**>,

`Ch`(*s*)=□, `Sw`(*s*)=*Off*, `SwitchOn`(*s*)=**N** if *length*(*w*) is even}.

For the final statement in the program,

`IF SwitchOn ...` =

(SwitchOn = 'Y' and OUTPUT=(*x*,††,**W**) → OUTPUT :=
<*x* & †**Input has odd length**†,††,**W**>) |

(SwitchOn <> 'Y' and OUTPUT=(*x*,††,**W**) → OUTPUT :=
<*x* & †**Input has even length**†,††,**W**>)

None of the previous functions has altered the value of OUTPUT, so the already-written portion *x* must be empty. Composing `IF SwitchOn ...` with the meaning of the rest of the BEGIN statement yields:

{<*r*, *t*>: where `INPUT`(*t*)=<*w*,††,**R**>, `Ch`(*t*)=□, `Sw`(*t*)=*On*,

`OUTPUT`(*t*)=<†**Input has odd length**†,††,**W**>,

`SwitchOn`(*t*)=**Y** if *length*(*w*) is odd} ∪

{<*r*, *t*>: where `INPUT`(*t*)=<*w*,††,**R**>, `Ch`(*t*)=□, `Sw`(*t*)=*Off*,

`OUTPUT`(*t*)=<†**Input has even length**†,††,**W**>,

`SwitchOn`(*t*)=**N** if *length*(*w*) is even}.

Applying:

`VAR Ch, SwitchOn: CHAR; Sw: Switch`^{T} ∘ `.`

gives exactly the same function as the comment for OddOrEven indicates.

18.3.4 Exercises

18.3.1

a) Give a formal representation function *A* for the `Rational` module of Section 18.2.1.

Show that the abstract comments for the following operations are correct, using the given concrete comment and the representation of a).

b)
```
PROCEDURE AddRat(VAR Result, Op1, Op2: Rational);
    {abs: Result := Op1 + Op2 }
    {con: Result.Den,Result.Num :=
            Op1.Den * Op2.Den,
            Op1.Num * Op2.Den + Op1.Den * Op2.Num }
```

c)
```
PROCEDURE Square(VAR R: Rational);
  {abs: (R<>0 --> R := R*R) | (R=0 --> )
   con: (R.Num*R.Den <> 0 --> R.Num,R.Den :=
            R.Num*R.Num, R.Den*R.Den ) |
        (R.Num*R.Den = 0 -->
            R.Num,R.Den := 0,1 ) }
```

d) If the module is changed to the alternate representation of Section 18.2.4, would it also alter a) - c) above? Explain.

18.3.2 Objects of type `Switch` (Section 18.3) could have been implemented as an integer rather than a character:

```
TYPE
  Switch = RECORD
             C: INTEGER
           END;
```

`Flip` could then have been written:

```
PROCEDURE Flip(VAR Sw: Switch);
  {abs: (Sw=On --> Sw := Off) |
        (Sw=Off --> Sw := On)
   con: (Sw.C=0 --> Sw.C := 1) |
        (Sw.C=1 --> Sw.C := 0) }
  BEGIN {Flip}
    Sw.C := (Sw.C+1) MOD 2
  END;  {Flip}
```

a) Show that the concrete comment is *not* a description of the procedure's actual statement, and replace it with a concrete comment that is correct.

b) Prove that the abstract comment is correct.

18.3.3 In the verification of `Switch` of Section 18.3.2:

a) Prove that the concrete comment correctly describes the action of the actual procedure statements for `Flip`.

b) Give the details of the proof that `Test` is correctly implemented.

c) Prove that the suggested function for the WHILE statement is correct.

18.3.4 Given the following heading:

```
PROCEDURE Push(VAR S: Stack; C: CHAR);
  { abs: S := S & <C>
    con: S.T=<,x,R> --> S.T := <,cx,R> }
```

and type declaration:

```
TYPE
  Stack = RECORD
            T: TEXT
          END;
```

give a representation function for `Stack` objects and show that the abstract comment is correct.

18.3.5 A portion of a module for the data abstraction `String` is given below.

```
CONST
  MaxL = 5;
TYPE
  String = RECORD
             C1, C2, C3, C4, C5: CHAR;
             Len: 0..MaxL
           END;
  {Strings are stored in the Ci variables beginning
   with C1 and continuing to the proper values for
   the length stored in Len.
   No string may be more than MaxL characters long.}
PROCEDURE CatString
  (VAR Result: String; First, Second: String);
  {abs: (length(First) + length(Second) <= MaxL -->
           Result := First & Second) |
        (length(First) + length(Second) > MaxL --> )}
  VAR
    Ctr: INTEGER; Tmp: CHAR;
  BEGIN
    IF First.Len + Second.Len <= MaxL
    THEN
      BEGIN
        Result := First;
        Result.Len := First.Len + Second.Len;
        FOR Ctr := 1 TO Second.Len
        DO
          BEGIN
            CASE Ctr
            OF
              1: Tmp := Second.C1;
              2: Tmp := Second.C2;
              3: Tmp := Second.C3;
              4: Tmp := Second.C4;
              5: Tmp := Second.C5
            END;
            CASE First.Len + Ctr
            OF
              1: Result.C1 := Tmp;
              2: Result.C2 := Tmp;
              3: Result.C3 := Tmp;
              4: Result.C4 := Tmp;
              5: Result.C5 := Tmp
            END
          END
      END
  END
```

a) Write a concrete comment describing the implementation of the operation `CatString`.

b) Give a representation function for this implementation of strings.

c) Verify the implementation.

18.3.6 Consider a procedure P without parameters and without any conflicts involving its local identifiers. P contains a comment that claims to give the box function of the body of the procedure.

a) Explain how it could be useful in calculating the meaning of the whole program, to first check that the comment in P is accurate in its claim.

b) Suppose that P is part of a data abstraction module, and has two comments, one for its abstract function and one for its concrete function. Explain how the verification of P (showing that the abstract comment is accurate) could be useful in calculating the meaning of a program using P. Distinguish this case from the one (a) that doesn't involve abstraction.

18.4 Chapter Summary

The abstract comments written for the operations of a module are all the external user need know about its meaning. If these comments correctly describe the actual module operations, they can be used in meaning calculations for any module use. The verification of a module therefore consists of a demonstration that each abstract comment is correctly implemented by the implementation procedure. Central to this proof is a formal representation mapping that establishes a correspondence between the concrete and abstract worlds. All possible implementation diagrams for the module commute if and only if the diagrams for each operation commute.

PART V: PROGRAM DESIGN IN O PASCAL

The advanced facilities of O (Optimization) Pascal permit random data references and execution control. Two new data types in O Pascal permit random data references, arrays and pointers. A variable of type:

ARRAY [*J*] OF *T*

is a bounded list of elements of type *T*, whose elements are indexed by values of an ordinal type *J*. For any type *T*, there is also a predefined, unbounded list of elements of type *T* indexed by values of an unordered type called "pointer."

Program parts with arrays or pointers have part functions, but the formal derivation of these part functions from the program text is beyond the scope of this book. Arrays and pointers can provide large speed improvements in a program, but they must be used with great care. They should be used only when their program part functions can be determined and verified at least informally. We have already seen a similar situation with iteration statements. Any terminating WHILE statement can be replaced by a group of nested IF statements, but the WHILE statement is so efficient (in terms of program text required), that the loops are worth the extra effort they require for the determination of their part functions. Indeed, there is no mechanical way to prove the termination of loops, and therefore no mechanical way to verify the part function of a WHILE statement. Nevertheless, with discipline, we can restrict ourselves to writing WHILE statements whose termination can be proved by *ad hoc* proofs. Similarly, even though a mechanical derivation of part functions may not be possible when arrays and pointers are used, we can restrict ourselves to uses in which *ad hoc* derivations and verifications are possible.

GOTO statements permit transfer of execution control between almost any statements of a program; their program part functions are also beyond the scope of this book. However, GOTO statements are only introduced for efficiency reasons. They may be systematically removed to obtain an equivalent GOTO-free program that can be more easily analyzed.

O Pascal contains a new data type, REAL, which approximates real numbers. Its advantage is that, up to wide bounds, numbers of any size can be treated uniformly. Its disadvantage is that numbers are rounded off in computation and round-off errors can accumulate to invalidate the results.

CHAPTER 19

AGGREGATE DATA TYPES IN O PASCAL

Chapter Preview

The array data type provides a collection of objects in which each member may be accessed as quickly as any other. The size of the collection is limited and inflexible, but when size restrictions are not important, arrays can provide a dramatic speed increase. Arrays are useful for implementing abstract data types and for operations involving searching and sorting.

Arrays are fixed-size collections of objects that can be used to replace either functions (when their domains are small) or files (when they contain only a few components). Like functions, arrays map elements from a domain of one type to a range of a possibly different type. With arrays simulating functions, the value of a function can be "looked up" instead of computed each time it is needed, often with a large saving in time. When an array simulates a file, access to any element takes a fixed amount of time, providing a large speedup when elements are needed in an unpredictable order.

19.1 Arrays

Preview

An array is formed from two types: its components and its indices. The index values are used to select component values. The power of arrays is in the ability to compute an index value, then use it to obtain a component value, in a time that does not depend on either value or the past history of array use.

Pascal array types are declared with a statement of the form:

```
TYPE
  ArrType = ARRAY [IndexType] OF ComponentType;
```

The domain of this array is `IndexType` and its range is a subset of `ComponentType`. In Pascal, `IndexType` is limited to ordinal types other than `INTEGER` so that the number of values in the domain is fixed by the declaration. An array variable, say `ArrVar1` or `ArrVar2`, can be declared to have this type, and a variable of the index type will be useful:

```
VAR
  Index: IndexType;
  ArrVar1, ArrVar2: ArrType;
```

Array variables map values of type `IndexType` to values of type `ComponentType`. This mapping is obtained by *subscripting* the array variable with an index value. The subscripting operation in Pascal is indicated by square brackets, e.g.:

```
ArrVar1[Index]
```

The mapping represented by an array is finite and explicit; each domain element is given its associated range element by an assignment. The assignment is usually done one index element at a time, but also an entire array can be assigned to another array. The statement:

```
ArrVar1[Index] := CTypeValue
```

changes the value of the mapping represented by `ArrVar1` at the point in the domain specified by `Index` to the value of `CTypeValue`, but leaves the remainder of the mapping unchanged. The statement:

```
ArrVar1 := ArrVar2
```

changes the entire mapping by assigning all the components of ArrVar2 to the corresponding components of ArrVar1.

In the following example, Arr has both of its types a subrange of INTEGER, and the FOR statement assigns each element in the array a value equal to its position. That is, this array is assigned the identity function on the domain {1, 2, 3, 4, 5}.

```
TYPE
  Both = 1..5;
VAR
  I: Both;
  Arr: ARRAY [Both] OF Both;
BEGIN
  FOR I := 1 TO 5
  DO
    Arr[I] := I
END
```

Code ciphers are an application ideally suited to the use of an array. The simplest way to encrypt a message is to use a substitution cipher that replaces the characters of a message with characters from a code. A message can be represented as an array of characters, with the index values integers that are the position of each character in the message. Suppose that messages of length up to Len are formed only of capital letters:

```
CONST
  Len = 20;
TYPE
  Str = ARRAY [1..Len] OF 'A'..'Z';
VAR
  Msg: Str;
```

Another array of characters can hold the cipher characters (not limited to letters), with index values the letters that occur in a message.

```
TYPE
  Cipher = ARRAY ['A'..'Z'] OF CHAR;
VAR
  Code: Cipher;
```

Characters in a message that are not capital letters are reproduced without substitution.

The following design part reads and encrypts an arbitrary number of lines of input according to the scheme described above. Since each line of input is stored in Msg prior to encryption, Msg is padded with blanks after each line is read so that characters read on a previous, longer line do not appear in the encrypted output of a subsequent, shorter line.

Design Part 1

```
PROGRAM Encryption(INPUT,OUTPUT);
  {translate the characters in INPUT according
   to the code in Cipher and print the new
   characters in OUTPUT}
  CONST
    Len = 20;
  TYPE
    Str = ARRAY [1..Len] OF 'A'..'Z';
    Cipher = ARRAY ['A'..'Z'] OF CHAR;
  VAR
    Msg: Str;
    Code: Cipher;
  BEGIN {Encryption}
    {initialize Code};
    WHILE NOT EOF
    DO
      BEGIN
        {read a line into Msg and echo it};
        {pad Msg with blanks};
        {print coded Msg}
      END
  END.  {Encryption}
```

Initializing the cipher array is a long series of assignment statements so it is packaged in a procedure.

Design Part 1.1

```
{initialize Code}
Initialize(Code);
```

Design Part 1.1.1

```
PROCEDURE Initialize(VAR Code: Cipher);
  {assign substitution cipher to Code}
  BEGIN {Initialize}
    Code['A'] := 'Z';
    Code['B'] := 'Y';
    Code['C'] := 'X';
    Code['D'] := '#';
    Code['E'] := 'V';
    Code['F'] := 'U';
    Code['G'] := 'T';
    Code['H'] := 'S';
    Code['I'] := 'I';
    Code['J'] := 'Q';
    Code['K'] := 'P';
    Code['L'] := '!';
    Code['M'] := 'N';
    Code['N'] := 'M';
    Code['O'] := '2';
    Code['P'] := 'K';
    Code['Q'] := '$';
    Code['R'] := 'D';
    Code['S'] := 'H';
    Code['T'] := '*';
    Code['U'] := 'F';
```

```
    Code['V'] := 'E';
    Code['W'] := 'T';
    Code['X'] := 'C';
    Code['Y'] := 'B';
    Code['Z'] := 'A'
  END;  {Initialize}
```

Characters are read into consecutive elements of `Msg` until a line mark is encountered.

Design Part 1.2

```
{read a line into Msg and echo it}
I := 0;
WHILE NOT EOLN AND (I < Len)
DO
  BEGIN
    I := I + 1;
    READ(Msg[I]);
    WRITE(Msg[I])
  END;
READLN;
WRITELN;
```

The index `I` is now positioned at the last character of the message (up to `Len`), and blanks should be added to pad out the line:

Design Part 1.3

```
{pad Msg with blanks}
FOR I := I+1 TO Len
DO
  BEGIN
    Msg[I] := ' '
  END;
```

Since translating `Msg` takes several lines of code, they are packaged in a procedure.

Design Part 1.4

```
{print coded Msg}
Encode(Msg)
```

`Msg` elements `A` to `Z` are used as subscripts into `Code` to obtain their translations. Other characters are printed without translation.

Design Part 1.4.1

```
PROCEDURE Encode(VAR S: Str);
  {write the characters in Code corresponding to
   the alphabetic characters in S}
  VAR
    Index: 1..Len;
  BEGIN {Encode}
    FOR Index := 1 TO Len
    DO
      IF S[Index] IN ['A'..'Z']
      THEN
        WRITE(Code[S[Index]])
      ELSE
        WRITE(S[Index]);
    WRITELN
  END;  {Encode}
```

The complete program and some sample input and output appear below.

```
PROGRAM Encryption(INPUT,OUTPUT);
  CONST
    Len = 20;
  TYPE
    Str = ARRAY [1..Len] OF 'A'..'Z';
    Cipher = ARRAY ['A'..'Z'] OF CHAR;
  VAR
    I: 1..Len;
    Msg: Str;
    Code: Cipher;
  {Include PROCEDURE Initialize(VAR Code: Cipher);}
  {Include PROCEDURE Encode(VAR S: Str);}
  BEGIN {Encryption}
    Initialize(Code);
    WHILE NOT EOF
    DO
      BEGIN
        I := 0;
        WHILE NOT EOLN AND (I < Len)
        DO
          BEGIN
            I := I + 1;
            READ(Msg[I]);
            WRITE(Msg[I])
          END;
        READLN;
        WRITELN;
        FOR I := I+1 TO Len
        DO
          BEGIN
            Msg[I] := ' '
          END;
        Encode(Msg)
      END
  END.  {Encryption}
```

Execution

```
INPUT :NOW IS THE TIME.
       END
OUTPUT:NOW IS THE TIME.
       M2T IH *SV *INV.
       END
       VM#
```

19.1.1 Syntax and Semantics for Arrays

The following syntax rules describe array types.

<aggregate type> ::= <unpacked structured type>

<unpacked structured type> ::= SET OF <base type>
| FILE OF <component type> | RECORD <field list> END
| ARRAY [<index type list>] OF <component type>

<index type list> ::= <index type list> , <ordinal type> | <ordinal type>

Since the values of array components can be referenced and defined like those of other variables, the definition of <variable> must be expanded to include them.

<variable access> ::= <variable identifier> | <buffer variable>
| <component variable>

<component variable> ::= <field designator> | <indexed variable>

<indexed variable> ::= <array variable> [<expression list>]

<array variable> ::= <variable access>

<expression list> ::= <expression list> , <expression> | <expression>

Formally, the values that correspond to array variables in a state are relations. The pairs of an array-value relation have their first members in the (finite) set of values of the index type, and their second members in the set of values of the component type. Thus the number of possible values of an array variable is large. Each index could be paired with any of the component values, so if there are N indices and M components possible, the array can have any of M^N values, each one a mapping from the indices to corresponding components. For example, the variable:

```
VAR
  CArray: ARRAY [1..3] OF CHAR;
```

has $128^3 \sim 2{,}100{,}000$ possible values for a Pascal machine with 128 distinct character values, one of which is {<1,**X**>, <2,**B**>, <3,**3**>}. This array value would result from the assignments:

```
CArray[1] := 'X';
CArray[2] := 'B';
CArray[3] := '3'
```

Once each index of an array has an assigned component, the array value must be a function. However, so long as any index remains unassigned, that part of the array acts like an uninitialized variable. For example, if only the first and third assignments above have been applied to `CArray`, its value is

{<1,**X**>, <3,**3**>} ∪ {<2,*c*>: *c* is a character},

which we abbreviate as usual to

{<1,**X**>, <3,**3**>, <2,?>}.

These values may occur in a state with the array identifier, e.g.,

S = {CArray·{<1,**X**>, <3,**3**>, <2,?>}, ...}.

In the special case that the entire array is uninitialized, the further abbreviation CArray·? will be used.

The meaning of an <indexed variable> can be obtained from the value of the corresponding array. For example, for the state S above,

$\boxed{\texttt{CArray[1]}}(S) = \mathbf{X}$.

In general, for array identifier A and index I, where I is of type J:

$\boxed{A\texttt{[}I\texttt{]}} = \{\langle s, v\rangle$: $s(I)$ is a value of type J, $v = (s(A))(s(I))\}$.

The definition of assignment (Section 6.2.1) applies to arrays without change, but does not cover the case where an array element stands on the left. In that case the assignment aborts unless the subscript has a proper value for its type. Formally, for a subscript I of type J:

$\boxed{A\ [\ I\ ] := E} = \{<s, t>: \boxed{I}(s)$ is a value of type J,

$t = s$ except that $\boxed{A\ [\ I\ ]}(t) = \boxed{E}(s)\}$.

In all other cases, the value of the expression becomes attached to the left-side identifier in the state following the assignment. When the assignment assigns one bare array identifier to another, this value is the relation that is the whole array value.

19.1.2 Array parameters

Entire arrays or components of arrays can appear as actual parameters in programs. As with Pascal's other aggregate types, arrays are usually bound to variable parameters for reasons of efficiency. If an array were passed to a value parameter, the implicit copying operation would have to duplicate every component. However, array components can be efficiently bound to either value or variable parameters. The definitions of Section 9.1.4 are not adequate to handle the case of aliasing between a subscript identifier and an array component with that subscript, passed to variable parameters. The following program indicates some of the complications that arise. `Index` is passed to `First` and `Arr[Index]` is passed to `Second`. Thus `Second` is aliased (through the subscript `Index`) to `First`. `First` (or `Index`) is incremented before `Second` (or `Arr[Index]`) is referenced. However, it is the value of `Arr[2]` rather than the value of `Arr[3]` that is altered because the binding of the name of the actual parameter to the formal parameter occurs when a procedure is called rather then when a variable parameter is referenced.

```
PROGRAM Bind(OUTPUT);
  VAR
    Arr: ARRAY [1 .. 5] OF INTEGER;
    Index: INTEGER;
  PROCEDURE IncreaseParms(VAR First, Second: INTEGER);
    BEGIN {IncreaseParms}
      First := First + 1;
      Second := Second + 1
    END; {IncreaseParms}
  BEGIN {Bind}
    FOR Index := 1 TO 5
    DO
      BEGIN
        Arr[Index] := 1
      END;
    Index := 2;
    IncreaseParms(Index, Arr[Index]);
    WRITELN('Index is', Index:2);
    FOR Index := 1 TO 5
    DO
      BEGIN
        WRITELN('Arr[', Index:1, '] is', Arr[Index]:2)
      END
  END. {Bind}
```

Execution

```
OUTPUT:Index is 3
       Arr[1] is 1
       Arr[2] is 2
       Arr[3] is 1
       Arr[4] is 1
       Arr[5] is 1
```

In this special case, when an array subscript and a component with that same subscript are passed as variable parameters, the actual parameter copied into the procedure body must have its subscript

evaluated in the state of the call, not a subsequent state within the body.

19.1.3 Exercises

19.1.1 Modify the program `Encryption` of Section 19.1.1 so that

a) the blanks between words are encoded, so a code breaker would not have this clue to go by.

b) instead of padding with blanks, the program keeps track of which characters in `Msg` are a valid part of the input, using a variable of type `1..Len` to record the subscript "high water mark."

19.1.2 Discuss the potential difficulty in modifying `Encryption` (Section 19.1.1) so that it can handle *arbitrarily* long lines. Is this a practical difficulty?

19.1.3 In the `Encryption` program of Section 19.1.1:

a) Assume that the contents of the array `Code` defines a one-to-one mapping; that is, no range element is duplicated. Modify `Encryption` so that using the *same* `Code` array (not one with the code inverted) it can *decode*. That is, the input is a coded message, and the output is to be the "clear" text.

b) Suppose that the contents of `Code` is not necessarily one-to-one. Write a procedure to check for this property and print the duplications that exist.

c) Write a decoding program using `Code` as it exists, to handle the case where there may be duplications in the range of `Code`. For each encrypted input, print all possible clear texts.

19.1.4 Which declarations and statements in the following Pascal fragment contain errors?

```
TYPE
  AType = ARRAY [0..4] OF BOOLEAN;
VAR
  I,J: INTEGER;
  A: ARRAY [I..J] OF INTEGER;
  B,C: ARRAY [0..4] OF BOOLEAN;
  D: AType;
  E: ARRAY [0..4] OF BOOLEAN;
  F: AType;
  G: ARRAY [1..5] OF BOOLEAN;
BEGIN
  IF B[I+J] = D[I*J]
  THEN
    WRITELN(C[PRED(0)]);
  B := C;
  B := D;
  B := E;
  D := F;
  E := G
END
```

19.1.5 Each Pascal fragment below contains the identifier `Qu`. You are to deduce a possible type of `Qu` from the context in each separate case. Express the answer as a type specification for the declaration:

```
VAR
  Qu: ?
```

Replace the question mark with the type required of `Qu`. For example, if you decide that one answer is a subrange of `CHAR` between **3** and **6**, fill in:

```
VAR
  Qu: '3'..'6'
```

If there are several possible types, explain briefly.

a) `2 * Qu['X']`

b) `Qu[Red] AND (SUCC(Red) = Green)`

c) `Qu['X'].Component = 7`

d) `(Qu[Qu[C]] + C > 0) OR (Qu[C] = 'X')`

19.1.6 Consider the Pascal fragment

```
VAR
  B, C: ARRAY [0..4] OF BOOLEAN;
  I: 0..4;
BEGIN
  B[0] := TRUE;
  FOR I := 1 to 4
  DO
    B[I] := NOT B[I-1];
  C[2] := FALSE
END
```

What execution state exists after the final assignment?

19.1.7 Suppose that the abbreviation

```
Arr·?
```

appears in an execution state. Write out a complete formal definition of its meaning.

19.1.8 Write out a special case of the definition of procedure-parameter meaning in Section 9.1.4 that applies to aliasing like that found in `Bind` (19.1.3).

19.2 Implementing Abstract Types with Arrays

Preview

A data abstraction in which the concrete objects are finite can be implemented very efficiently using arrays.

The array data type implements finite lists with efficient, random access to any list member. The file type provides access to any member of a file, of course, but only in sequential order. If the desired member is not near the head of the file's future list when it is needed, many READ statements will be required to obtain it.

19.2.1 Finite TEXT Files

Arrays can be used directly to implement a finite version of CF Pascal text files. The abstract type `SmallFile` consists of the declaration:

```
CONST
  MaxFileLen = 100;
TYPE
  SmallFile = RECORD
                Str: ARRAY[0..MaxFileLen] OF CHAR;
                Cursor: 0..MaxFileLen;
                Mode: (Reading, Writing)
              END;
```

and procedures with the headings:

```
PROCEDURE InitializeSF(VAR F: SmallFile);
PROCEDURE ResetSF(VAR F: SmallFile);
PROCEDURE RewriteSF(VAR F: SmallFile);
PROCEDURE ReadSF(VAR F: SmallFile; VAR Ch: CHAR);
PROCEDURE WriteSF(VAR F: SmallFile; Ch: CHAR);
```

Each procedure can be used in place of the corresponding file operation. For example, the sequence of statements

```
RESET(F);
READ(F, Ch)
```

for a new TEXT file F and CHAR Ch could be replaced by

```
InitializeSF(Sf);
ResetSF(Sf);
ReadSF(Sf, Ch)
```

using a SmallFile Sf. So long as the MaxFileLen limit is observed, there should be no difference except that the SmallFile operations will be considerably faster on most Pascal machines.

The ResetSF and ReadSF procedures are:

```
PROCEDURE ResetSF(VAR F: SmallFile);
  BEGIN {ResetSF}
    IF F.Mode=Writing
    THEN {fill with blanks}
      FOR F.Cursor := F.Cursor TO MaxFileLen
      DO
        F.Str[F.Cursor] := ' ';
    F.Cursor := 0;
    F.Mode := Reading
  END;  {ResetSF}

PROCEDURE ReadSF(VAR F: SmallFile; VAR Ch: CHAR);
  BEGIN {ReadSF}
    IF (F.Mode=Reading) AND (F.Cursor<MaxFileLen)
    THEN {file still readable}
      BEGIN
        Ch := F.Str[F.Cursor];
        F.Cursor := F.Cursor + 1
      END
    ELSE {undefined}
      WRITELN('Undefined operation')
  END;  {ReadSF}
```

The treatment of end file is not the same as for TEXT files, but the difference will not matter so long as a program observes the discipline of reading no more than was previously written in a SmallFile.

19.2.2 Finite Stacks

A bounded stack can be implemented with an array and an integer. The integer is zero if the stack is empty; otherwise it is the subscript of the top element of the stack, the others being stored below it down to subscript 1. Since there are only a finite number of array components, only a finite number of values can be pushed on a stack implemented in this way.

In the following example, the first Depth characters in the input are reversed by pushing their values on a stack, then printing and popping the top stack value.

```
PROGRAM Reverse(INPUT,OUTPUT);
  {reverse the first Depth characters in INPUT}
  CONST
    Depth = 20;
  TYPE
    EltType = CHAR;
    Stack = RECORD
              Val: ARRAY [1..Depth] OF EltType;
              StackTop: 0..Depth
            END;
  VAR
    S: Stack;
    Elt: EltType;
  PROCEDURE NewStack(VAR S: Stack);
    {S := <>}
    BEGIN {InitStack}
      S.StackTop := 0
    END;  {InitStack}
  PROCEDURE Push(VAR S: Stack; E: EltType);
    {S := S & <E>}
    BEGIN {Push}
      IF S.StackTop >= Depth
      THEN
        WRITELN('** OVERFLOW **')
      ELSE
        BEGIN
          S.StackTop := S.StackTop + 1;
          S.Val[S.StackTop] := E
        END
    END;  {Push}
  PROCEDURE Pop(VAR S: Stack);
    {There are Stack X, EltType E such that
     S = X & <E> --> S := X}
    BEGIN {Pop}
      IF S.StackTop <= 0
      THEN
         WRITELN('** UNDERFLOW **')
      ELSE
         S.StackTop := S.StackTop - 1
    END;  {Pop}
```

```
FUNCTION Top(VAR S: Stack): EltType;
  {There is a Stack X, EltType E such that
   S = X & <E> --> Top := E}
  BEGIN {Top}
    IF S.StackTop <= 0
    THEN
      BEGIN
        WRITELN('** READING EMPTY STACK **');
        Top := '0'
      END
    ELSE
      Top := S.Val[S.StackTop]
  END;  {Top}
FUNCTION Empty(VAR S: Stack): BOOLEAN;
  {Empty := (S = <>)}
  BEGIN {Empty}
    Empty := S.StackTop <= 0
  END;  {Empty}
FUNCTION Full(VAR S: Stack): BOOLEAN;
  {Full := (Length(S) = Depth)}
  BEGIN {Full}
    Full := S.StackTop >= Depth
  END;  {Full}
BEGIN {Reverse}
  NewStack(S);
  WHILE NOT EOF AND NOT Full(S)
  DO
    BEGIN
      READ(Elt);
      Push(S,Elt)
    END;
  WHILE NOT Empty(S)
  DO
    BEGIN
      WRITE(Top(S));
      Pop(S)
    END;
  WRITELN
END.  {Reverse}
```

Execution

```
INPUT :abcdefghijklmnopqrstuvwxyz
OUTPUT:tsrqponmlkjihgfedcba
```

Finite TEXT files and bounded stacks are two abstract types that could become the basis for a library of types.

19.2.3 Exercises

19.2.1 Explain why the InitializeSF operation is necessary, in addition to ResetSF and RewriteSF, for the SmallFile data abstraction of Section 19.2.1.

19.2.2 Add abstract comments to the procedure headers of the SmallFile data abstraction of Section 19.2.1.

19.2.3 The implementation of `SmallFile` is limited in the number of items that may be placed in the data structure.

a) Write an exact description of how this limitation shows up in the given `ReadSF`, and how it should enter in the procedure `WriteSF`. What happens when the limits are exceeded?

b) Do the implementation for `RewriteSF` and `WriteSF`.

19.2.4 Suppose that a `SmallFile` has had exactly `MaxFileLen` `WriteSF` operations performed on it. Explain why later reading this file with `ReadSF` does not behave as would a combination of WRITE and READ statements on an actual `TEXT` file.

19.2.5 Implement an improved `SmallFile` data abstraction by adding an analog of the `TEXT` file `EOF` function, and an `EOM` (for End Of Medium) function that can be used before `WriteSF` to see if in fact a write operation is possible, or if the implementation limits have been exceeded.

19.2.6

a) Isolate the `Stack` data abstraction within the program `Reverse` of Section 19.2.2.

b) Can the `Stack` components be changed to *any* desired type by just replacing the `TYPE` declaration for `EltType`? Explain.

19.2.7

a) Are the comments for `Stack` (Section 19.2.2) abstract or concrete? Explain.

b) Give a representation function for `Stack`.

c) Verify the `Push` operation of `Stack`.

19.2.8 Solve the Towers of Hanoi problem described in Chapter 6 of the text but without using recursion. Your solution should contain the data types `Move` and `StackOfMoves`. Each `Move` should be represented by a record containing information about the number of disks to be moved, the origin of the move, etc. The `StackOfMoves` should be represented as an array and a top of stack index. Moves of more than a single disk must be broken into smaller moves and pushed on the stack until they can be attempted. For example, to move n disks from peg A to peg B using peg C as intermediate storage, first move n-1 disks from A to C, moving a single disk from A to B, and moving n-1 disks from C to B. As a sample, show the behavior of the program when moving 3 disks from peg A to peg B.

19.3 Searching and Sorting Arrays

Preview

Because array elements may be accessed in any order with no time penalty, new methods of sorting and searching become available.

Sequences of values can be stored in arrays as well as in files. The major differences between files and arrays are the number of components and the ability to access components. The number of components in a file is not specified by the programmer, but grows during execution time to a limit imposed by the Pascal machine, while the number of components of an array is defined in the declaration of the array and cannot be changed during execution. Each component of an array is immediately accessible, but only a single component of a file (the first from the future list) is. A file has a built-in mechanism (`EOF`) for detecting its end, but only part of an array may contain valid values, and this must be explicitly provided for by the programmer.

19.3.1 Searching an Array

An array can be searched as a file is: by examining each component in order. However, when not all components of the array should participate, a way must be found to mark the end of valid data in the array. A clever way is to use the element to be found as a *sentinel*, placing it in the array at the end to stop the search.

Design Part 2

```
PROGRAM SentinelSearch (INPUT, OUTPUT);
  {INPUT contains integer values:
   Key, i1, ..., iN, N=ArrSize-1 -->
    ((There is a J, 1<=J<ArrSize, iJ=Key -->
        OUTPUT := Key & 'found at position J') |
     (For all J, 1<=J<ArrSize, iJ<>Key -->
        OUTPUT := Key & 'not found')) }
  CONST
    ArrSize = 11;
  VAR
    Arr: ARRAY [1 ..  ArrSize] OF INTEGER;
    Index: INTEGER;
  BEGIN {SentinelSearch}
    {get values for Arr and Key};
    Index := 1;
    WHILE (Index <= ArrSize) AND
          (Arr[ArrSize] <> Arr[Index])
    DO
      Index := Index + 1;
    IF Index < ArrSize
    THEN
      WRITELN(Arr[ArrSize], ' found at position', Index)
    ELSE
      WRITELN(Arr[ArrSize], ' not found')
  END.
```

The value to be found is read directly into the last position of `Arr`:

Design Part 2.1

```
{get values for Arr and Key}
READ(Arr[ArrSize]);
FOR Index := 1 TO ArrSize-1
DO
  READ(Arr[Index]);
```

Assembling these two parts:

Execution

```
INPUT: 9 0 1 2 3 4 5 9 8 7 6
OUTPUT:          9 found at position          7
```

`SentinelSearch` takes no advantage of an array's ability to access components out of order. A technique called *binary search* can improve the execution speed of the search if the data list is sorted. The idea is to probe into the list at the halfway point, discover on which side the sought value lies, then probe that side at its halfway point, and so on until only one possible position remains. The desired value must lie there or it is not in the array. For example, consider searching the following array for the value 70:

Pos	1	2	3	4	5	6	7
Arr[Pos]	10	20	30	40	50	60	70
Probes				*p1*		*p2*	*p3*

The first probe is made at position 4, the midpoint of the entire array. Since `Arr[4]` is less than the value sought, the lower portion of the array is eliminated and the next probe is made at 6, halfway through the upper half of the array. Again the lower portion is eliminated, leaving only `Arr[7]` to be examined. Thus three comparisons were necessary to search an array of seven values. In general for an array with 2^n-1 elements, the elements can be split in half at most n times before reaching a subsequence of the array containing only a single element. Thus instead of having to make at most 2^n-1 comparisons to locate a value, no more than $\log_2(2^n) = n$ comparisons are required.

Design Part 3

```
PROGRAM BinarySearch (INPUT, OUTPUT);
  {INPUT contains integer values:
   Key, i1, ..., iN, N=ArrSize;
   AND (For all K, 1<=K<ArrSize, iK <= i(K+1) ) AND
   (i1<=Key<=iArrSize) -->
    ((There is a J, 1<=J<=ArrSize, iJ=Key -->
        OUTPUT := Key & 'found at position J') |
     (For all J, 1<=J<=ArrSize, iJ<>Key -->
        OUTPUT := Key & 'not found')) }
  CONST
    ArrSize = 7;
  VAR
    Arr: ARRAY [1 ..  ArrSize] OF INTEGER;
    Low, High, Key: INTEGER;
  BEGIN {BinarySearch}
    {get values for Arr and Key};
    Low := 1;
    High := ArrSize;
    IF (Key >= Arr[Low]) AND (Key <= Arr[High])
    THEN
      {Search Arr[Low] to Arr[High] for Key}
    ELSE
      WRITELN(Key, ' not in range of values')
  END.  {BinarySearch}
```

Design Part 3.1

```
{get values for Arr and Key}
READ(Key);
FOR Index := 1 TO ArrSize
DO
  READ(Arr[Index]);
```

The binary search portion repeatedly divides the interval between `Low` and `High` in half until at most a single element remains. The value of this element is compared to that of `Key`.

Design Part 3.2

```
{Search Arr[Low] to Arr[High] for Key}
BEGIN
  WHILE High > Low
  DO
    {Divide interval until at most one
     element remains};
  IF Arr[Low] = Key
  THEN
    WRITELN(Key, ' found at position', Low)
  ELSE
    WRITELN(Key, ' not found')
END
```

The integer variable `Mid` is introduced to calculate the interval's midpoint. If the element at `Mid` has the same value as `Key`, the search is terminated by reducing the length of the interval to one. Otherwise, the search continues in the half of the interval that could possibly contain the `Key` value.

Design Part 3.2.1

```
{Divide interval until at most one element remains}
BEGIN
  Mid := (High + Low) DIV 2;
  IF Key = Arr[Mid]
  THEN {found - reduce interval to length one}
    BEGIN
      Low := Mid;
      High := Mid
    END
  ELSE {still looking - divide interval in half}
    IF Key > Arr[Mid]
    THEN
      Low := Mid + 1
    ELSE { Key < Arr[Mid] }
      High := Mid - 1
END;
```

Some sample executions are shown below:

Execution

```
INPUT: 70 10 20 30 40 50 60 70
OUTPUT:         70 found at position         7
INPUT: 5 10 20 30 40 50 60 70
OUTPUT:          5 not in range of values
INPUT: 33 10 20 30 40 50 60 70
OUTPUT:         33 not found
```

19.3.2 Sorting Arrays

In previous chapters characters in `TEXT` files were sorted. The algorithms were constrained by the sequential access and retrieval operations available for files. The sorting algorithms presented below take advantage of the random access operations supported by arrays.

The simplest method of sorting an array into ascending order is called an *exchange sort.* The smallest value is found and exchanged with the value in the first position. Now the array is sorted through the first element, and the same technique is applied to the remainder of the array.

Design Part 4

```
PROCEDURE ExchangeSort(VAR Vals: IntArray; Start, Size: INTEGER);
  VAR
    Min, Index, Save, Last: INTEGER;
  BEGIN {ExchangeSort}
    Last := Start+Size-1;
    FOR Start := Start TO Last
    DO
      BEGIN
        {Vals[Min] is the smallest value in
           [Vals[Start],...,Vals[Last]]};
        {exchange Vals[Start] and Vals[Min]}
      END
  END;  {ExchangeSort}
```

Design Part 4.1

```
{Vals[Min] is the smallest value in
   [Vals[Start],...,Vals[Last]]};
Min := Start;
FOR Index := Start+1 TO Last
DO
  IF Vals[Index] < Vals[Min]
  THEN
    Min := Index;
```

Design Part 4.2

```
{exchange Vals[Start] and Vals[Min]}
Save := Vals[Start];
Vals[Start] := Vals[Min];
Vals[Min] := Save
```

While this technique is easy to implement, it is not very fast. Suppose the value of `Size` is N. The first execution of the inner FOR statement requires N-1 comparisons to find the smallest element. Since the first element is not reexamined, the next execution of the FOR statement requires N-2 comparisons to find the smallest remaining element. The last execution of the FOR statement involves only a single element and requires N-N or zero comparisons since it must be the smallest remaining element. Thus, the total number of comparisons is:

$$N\text{-}1 + N\text{-}2 + \ldots + 0 = N^2 - (1+2+\ldots+N) = N^2 - N(N+1)/2 = N(N\text{-}1)/2.$$

The exchange-sort algorithm can be thought of as dividing the array into a sorted part (containing just the first element) and a further unsorted part. N divisions are required for an array for size N. An algorithm called *merge sorting* divides the array at the midpoint, sorts each piece, and merges the sorted pieces. For an array of size $N = 2^k$ for some k, the array can be divided at most $\log_2 N = k$ times before reaching a piece of size one which is obviously sorted. This algorithm lends itself to recursive implementation:

Design Part 5

```
PROCEDURE MergeSort(VAR Vals: IntArray;
                    Start, Size: INTEGER);
  BEGIN {MergeSort}
    IF Size > 1
    THEN
      BEGIN
        {sort the values in the first part of Vals}
        MergeSort(Vals,Start,Size DIV 2);
        {sort the values in the second part of Vals}
        MergeSort(Vals,Start+Size DIV 2,Size DIV 2);
        {merge the sorted halves}
        Merge(Vals,Start,Size)
      END
  END;  {MergeSort}
```

To merge two sorted subarrays, their first elements are compared and the smaller copied into a temporary array. The process is then repeated with the copied element removed. When the elements of one array are exhausted, remaining values from the other are added to the temporary array. Finally, the values of the temporary array are copied into the space originally occupied by both subarrays.

Design Part 5.1

```
PROCEDURE Merge(VAR Vals: IntArray;
                Start, Size: INTEGER);
  VAR
    FirstStart, FirstEnd,
    SecondStart, SecondEnd: INTEGER;
    Index: INTEGER;
    Temp: IntArray;
  BEGIN {Merge}
    {Divide Vals in half: FirstStart..FirstEnd
     and SecondStart..SeccondEnd}
    FirstStart := Start;
    SecondStart := Start + Size DIV 2;
    FirstEnd := SecondStart - 1;
    SecondEnd := Start + Size - 1;
    Index := 1;
    WHILE (FirstStart <= FirstEnd) AND
          (SecondStart <= SecondEnd)
    DO
      BEGIN
        {(Vals[FirstStart] < Vals[SecondStart] -->
           Temp[Index], FirstStart :=
             Vals[FirstStart], FirstStart+1) |
         (Vals[FirstStart] >= Vals[SecondStart] -->
           Temp[Index], SecondStart :=
             Vals[SecondStart], SecondStart+1) }
        Index := Index + 1
      END;
    {copy Vals[FirstStart] through Vals[FirstEnd] to Temp};
    {copy Vals[SecondStart] through Vals[SecondEnd] to Temp};
    {copy Temp to Vals[Start] through Vals[Start+Size-1]}
  END;  {Merge}
```

Design Part 5.1.1

```
{(Vals[FirstStart] < Vals[SecondStart] -->
   Temp[Index],FirstStart := Vals[FirstStart],FirstStart+1) |
 (Vals[FirstStart] >= Vals[SecondStart] -->
   Temp[Index],SecondStart := Vals[SecondStart],SecondStart+1)}
IF Vals[FirstStart] < Vals[SecondStart]
THEN
  BEGIN
    Temp[Index] := Vals[FirstStart];
    FirstStart := FirstStart + 1
  END
ELSE
  BEGIN
    Temp[Index] := Vals[SecondStart];
    SecondStart := SecondStart + 1
  END;
```

Design Part 5.1.2

```
{copy Vals[FirstStart] through Vals[FirstEnd] to Temp}
FOR FirstStart := FirstStart TO FirstEnd
DO
  BEGIN
    Temp[Index] := Vals[FirstStart];
    Index := Index + 1
  END;
```

Design Part 5.1.3

```
{copy Vals[SecondStart] through Vals[SecondEnd] to Temp}
FOR SecondStart := SecondStart TO SecondEnd
DO
  BEGIN
    Temp[Index] := Vals[SecondStart];
    Index := Index + 1
  END;
```

Design Part 5.1.4

```
{copy Temp to Vals[Start] through Vals[Start+Size-1]}
FirstStart := Start;
FOR Index := 1 TO Size
DO
  BEGIN
    Vals[FirstStart] := Temp[Index];
    FirstStart := FirstStart + 1
  END
```

Suppose that the array to be sorted is of size $N = 2^k$. Merging two sorted arrays of size $m/2$ each requires at most $m/2+m/2\text{-}1 = m\text{-}1$ comparisons since the last value is the largest and does not need to be compared to any other value. The following table shows the operations at each step, beginning with the deepest level of recursion:

Step	Merges	Array Size	Total Comparisons
1	$N/2$	1	$(N/2)(1)$
2	$N/4$	2	$(N/4)(3)$
3	$N/8$	4	$(N/8)(7)$
...	...	...	...
k-1	2	$N/4$	$2(N/2\text{-}1)$
$k=\log_2(N)$	1	$N/2$	$1(N\text{-}1)=N(N\text{-}1)/N$

Summing the total comparisons at each step gives the total number of comparisons for the entire sort:

$$N(\frac{1}{2}+\frac{3}{4}+\frac{7}{8}+...+\frac{N-1}{N}) .$$

Let $T = (\frac{1}{2}+\frac{3}{4}+\frac{7}{8}+...+\frac{N-1}{N})$. Then $2T = (1+\frac{3}{2}+\frac{7}{4}+...+\frac{2(N-1)}{N})$. Performing the subtraction $2T$ - T in the following peculiar way:

$$\begin{array}{rccccccccc} 2T = & 1 & + & 3/2 & + & 7/4 & + & ... & + & 0 \\ -T = & 0 & + & 1/2 & + & 3/4 & + & ... & + & (N\text{-}1)/N \\ \hline T = & 1 & + & 1 & + & 1 & + & ... & - & (N\text{-}1)/N \end{array}$$

gives

$$T = k - (N\text{-}1)/N = \log_2 N - (N\text{-}1)/N.$$

Thus the number of comparisons for the sort is

$$NT = N \log_2 N - N + 1 = N(k\text{-}1) + 1.$$

The following table shows the superiority of `MergeSort` to `ExchangeSort`:

Size N	k	Comparisons `ExchangeSort`	Comparisons `MergeSort`
4	2	6	5
8	3	28	17
64	6	2,016	321
1024	10	523,776	9,213

19.3.3 Exercises

19.3.1 Write a recursive version of `BinarySearch` (Section 19.3.1), using the fact that once the search array is reduced in size, searching it is an instance of the same problem.

19.3.2 An *insertion sort* is a good technique to use when data values must be placed in order as they arrive. An array holds the values received so far, in sorted order. When a new value arrives, its proper position in the array is found and it is inserted, moving the existing values out of the way. Write an interactive program that prompts for an integer as input, keeps the integers in an array sorted by insertion sort, and can print the array on demand.

19.3.3 A pseudo-key called a *hash code* can be computed from a search key, and used as an index to build or search a table. As long as all search keys hash to unique codes, a single comparison (to see if a table entry exists) will determine if the table contains a value equal to the search key. If a table is being built, the search key can be inserted at the position selected by its hash code. Many different algorithms exist for computing hash codes, e.g., for arrays of characters we might add the ordinal representation of the first several characters and use the sum modulo the table size as the hash code index.

Unfortunately different search keys can have identical hash codes, resulting in *collisions*. Keys whose hash codes collide may be stored elsewhere in the table. Common collision resolution techniques are storing the key in the next table location, or some fixed number of locations past the index location. For large, sparse tables hashing provides the most effective retrieval.

Write a program that builds and searches a hash table containing integers. Some care is required to determine when to stop searching for entries when collisions occur.

19.3.4 The sorting techniques in this section are based upon arbitrary values. Knowledge of the range of values can be used to decrease the time needed to sort them. To *bucket sort* a set of k possible values, an array with k index values (the "buckets") is created. A new value is inserted by increasing the count in its bucket. A sorted list (including duplicates) can be printed using the index values of the buckets. Write a program that bucket sorts examination scores, where the only possible scores are multiples of 5 between 50 and 100.

19.4 Array Extensions and Applications

Preview

PACKED arrays are stored so that they occupy less space in the Pascal machine. Multidimensional arrays are useful for storing tables and matrices.

19.4.1 The PACKED Attribute

The PACKED attribute can be applied to any aggregate type in Pascal.

<new structured type> ::= <unpacked structured type>
    ¦ PACKED <unpacked structured type>

Objects declared PACKED are stored in the Pascal machine so as to occupy minimum space, even if this results in longer processing time. Arrays of CHAR are the most frequently PACKED. Often, the programmer can use both kinds of arrays:

```
VAR
  PWord: PACKED ARRAY[PLb..PUb] OF CHAR;
  Word: ARRAY[Lb..Ub] OF CHAR;
```

With two arrays containing the same data, the programmer can exploit the tradeoff between execution time and storage space. Accessing packed array elements is relatively expensive (compared to accessing unpacked array elements) and assigning one packed array to another is relatively cheap (compared to assigning one unpacked array to another).

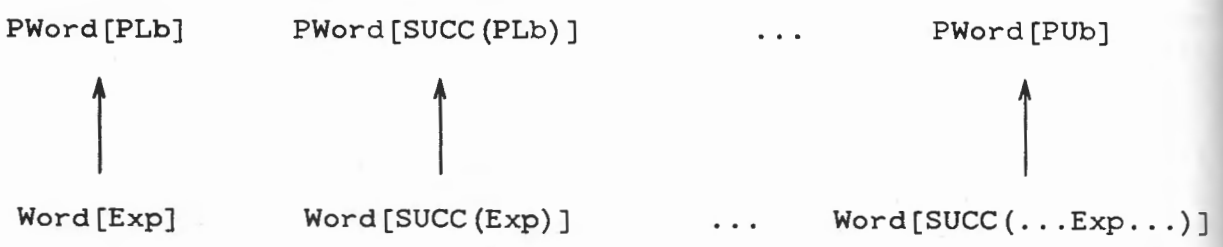
The operations PACK and UNPACK permit data to be moved from a variable of one representation to a variable of the other representation. For either of these procedures to be defined the size of the unpacked array must be at least as great as the size of the packed array. The PACK procedure moves values from an unpacked array to a packed array. With the declarations above, the statement

```
PACK(Word,Exp,PWord)
```

acts like:

```
FOR J := PLb TO PUb
DO
  PWord[J] := Word[J-PLb+Exp]
```

That is, the number of elements moved is the size of the packed array; the move begins at subscript Exp of the unpacked array. The components moved are:



The number of SUCC applications in the last subscript is the size of the unpacked array.

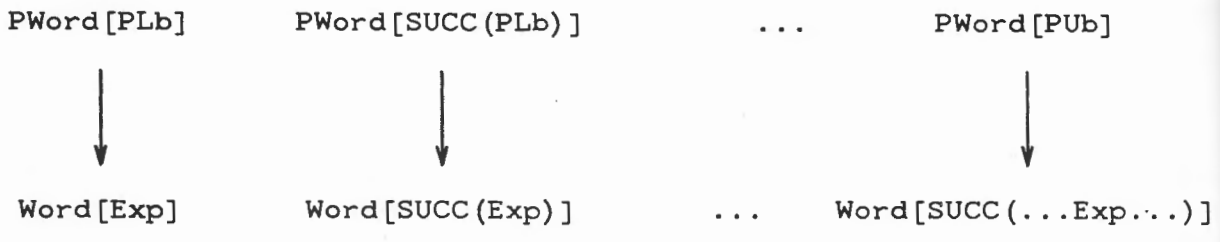
Similarly, the UNPACK procedure moves values from a packed array to an unpacked array. For the declarations above,

```
UNPACK(PWord,Word,Exp)
```

acts like:

```
FOR J := PLb TO PUb
DO
  Word[J-PLb+Exp] := PWord[J]
```

The number of components unpacked is the size of the packed array. The first component of the packed array is moved to the unpacked array at subscript Exp, and so on.



A special form of packed data has already been seen in CF Pascal: strings. A string constant of length StringLength is treated as if it has a type:

```
PACKED ARRAY[1..StringLength] OF CHAR
```

Thus the type of 'HELLO' is

```
PACKED ARRAY[1..5] OF CHAR
```

while that of 'GOODBYE' is:

```
PACKED ARRAY[1..7] OF CHAR
```

Just as these string constants can be written to TEXT files, so can any PACKED ARRAY ... OF CHAR. However, although it might seem sensible, strings may not be read from TEXT files. The relational operators can also be applied to strings, but only if the string lengths are the same. Thus, 'HELLO' and 'GOODBYE' cannot be compared because their lengths are not the same, but string 'HELLO ' can be compared with 'GOODBYE' since both have length 7. The comparison is the usual lexicographic one: characters are compared from the left, and the first different pair determines which string precedes the other, using the collating sequence for the Pascal machine. For example,

Expression	has value
'HELLO ' < 'GOODBYE'	FALSE
' HELLO ' <> 'HELLO '	TRUE

Components of packed arrays can only be bound to value parameters. Thus, for example, a single-character element from a packed array of characters may not be passed to a procedure with variable parameter of type character.

As an illustration of the use of packed and unpacked arrays, a program will be designed to print in lexicographic order Interrupt The program stores information in packed form (as an array of strings), but for processing unpacks the data one string at a time. When it comes time to print the results, the packed form can be used directly.

Design Part 6

```
PROGRAM Index(INPUT,OUTPUT);
  {create an index of the words in INPUT}
  CONST
    TableSize = 100;
    MaxWordLength = 20;
  TYPE
    WordType = PACKED ARRAY[1..MaxWordLength] OF CHAR;
    TableType =
      RECORD
        Word: ARRAY[1..TableSize] OF WordType; {word array}
        Entries: 0..TableSize {count of words in table}
      END;
  VAR
    Table: TableType;
    WordCount: 1..TableSize;
  BEGIN {Index}
    Table.Entries := 0; {empty table}
    {read and insert words in Table};
    {print words in Table};
    FOR WordCount := 1 TO Table.Entries
    DO
      WRITELN(Table.Word[WordCount])
  END {Index}.
```

Reading and inserting words in the table are both complex enough operations that they should be encapsulated in procedures. ReadWord will have two output parameters: Word (a string containing the text of the next word (if any) in INPUT, and Found (a BOOLEAN indicating whether or not a word was present).

Design Part 6.1

```
{read and insert words in Table};
ReadWord(INPUT,Word,Found);
WHILE Found
DO
  BEGIN
    InsertWord(Word,Table);
    ReadWord(INPUT,Word,Found)
  END;
```

ReadWord stores characters in an unpacked array, then packs them into the Result parameter. Short words are padded with blanks; excess characters in the input are discarded.

Design Part 6.1.1

```
PROCEDURE ReadWord(VAR FIn: TEXT; VAR Result: WordType;
  VAR Success: BOOLEAN);
  {ReadWord reads the next word from FIn, places the
   result in Result, and sets Success to indicate
   whether or not a word was read}
  VAR
    WideWord: ARRAY[1..MaxWordLength] OF CHAR;
    Ch: CHAR;
  BEGIN {ReadWord}
    Ch := ' ';
    WHILE NOT (EOF(FIn) OR (Ch IN ['A'..'Z','a'..'z']))
    DO {skip characters not in words}
      READ(FIn,Ch);
    IF NOT EOF(FIn)
    THEN
      BEGIN
        {read word, putting first MaxWordLength chars
         in WideWord};
        {pad WideWord with blanks};
        PACK(WideWord,1,Result);
        Success := TRUE
      END
    ELSE
      Success := FALSE
  END; {ReadWord}
```

Care must be taken in the next design part because end of file may be encountered while reading characters into `WideWord`.

Design Part 6.1.1.1

```
{read word, putting first MaxWordLength chars
 in WideWord}
ChCount := 0;
WHILE NOT EOF(FIn) AND (Ch IN ['A'..'Z','a'..'z'])
DO
  BEGIN
    IF ChCount < MaxWordLength
    THEN
      BEGIN
        ChCount := ChCount + 1;
        WideWord[ChCount] := Ch
      END;
    READ(FIn,Ch)
  END;
```

`WideWord` is padded with blanks before packing it into `Result`.

Design Part 6.1.1.2

```
{pad WideWord with blanks}
FOR ChCount := ChCount+1 TO MaxWordLength
DO
  WideWord[ChCount] := ' '
```

If `Word` is not in `Table`, `InsertWord` installs it in lexicographic order, using an insertion sort. Initially `Word` is placed at the end of `Table` so a sentinel search can be used.

Design Part 6.1.2

```
PROCEDURE InsertWord(VAR Word: WordType;
                     VAR Table: TableType);
  {Place Word in lexicographic order (no duplicates)
   in Table}
  VAR
    Index: 1..TableSize;
  BEGIN {InsertWord}
    IF Table.Entries < TableSize
    THEN
      BEGIN
        Table.Word[Table.Entries+1] := Word;
        WordCount := 1;
        WHILE Table.Word[WordCount] < Word
        DO
          WordCount := WordCount+1;
        {install Word}
      END
  END {InsertWord};
```

When the search terminates (as it must since `Word` was placed in the last entry of `Table`), `Table.Word[WordCount]` is lexicographically greater than or equal to `Word`. If `WordCount` indicates that `Word` matched the last `Table` entry, `Word` is right where it should be alphabetically and the number of table entries should be updated to indicate that a new word was found. If `WordCount` indicates that an earlier table entry is greater than or equal to `Word`, then either `Word` is already in the table (and no processing needs be done) or `Word` belongs within the table and the following entries have to be shifted down to make room to install it.

Design Part 6.1.2.1

```
{install word}
IF WordCount = Table.Entries+1
THEN {new word, greater than all other words}
  Table.Entries := WordCount
ELSE {Table.Word[WordCount] >= Word}
  IF Table.Word[WordCount] <> Word
  THEN {new word}
    BEGIN
      Index := WordCount;
      Table.Entries := Table.Entries + 1;
      WordCount := Table.Entries;
      WHILE WordCount >= Index+1
      DO {shift existing entries lower in table}
        BEGIN
          Table.Word[WordCount] :=
            Table.Word[WordCount-1];
          WordCount := WordCount - 1
        END;
      {install new word}
      Table.Word[Index] := Word
    END
```

After assembling the program and adding the declarations required by the variables introduced during development, the following sample execution occurs.

```
Execution
INPUT: WHILE Ch IN ['A'..'Z','a'..'z']
       DO
         BEGIN
           IF ChCount < MaxWordLength
           THEN
             BEGIN
               ChCount := SUCC(ChCount);
               WideWord[ChCount] := Ch
             END;
           IF EOF
           THEN
             Ch := ' '
           ELSE
             READ(Ch)
         END;
OUTPUT:A
       BEGIN
       Ch
       ChCount
       DO
       ELSE
       END
       EOF
       IF
       IN
       MaxWordLength
       READ
       SUCC
       THEN
       WHILE
       WideWord
       Z
       a
       z
```

19.4.2 Multidimensional Arrays

The range of an array can be any type, including another array type. In `Index` of the previous section, the table contained an array of packed array, and the range elements were accessed as complete arrays. The more common case is to use each component separately. Consider the declaration:

```
TYPE
  Matrix = ARRAY [1 .. 3] OF ARRAY [1 .. 5] OF INTEGER
```

A component of a `Matrix` variable `MArr` is selected by using two subscripts:

```
VAR
  MArr: Matrix;
  Index1: 1 .. 3;
  Index2: 1 .. 5;
BEGIN
  ...  MArr[Index1][Index2]...
END
```

The first subscript (`Index1`) selects one of three arrays, and the second subscript (`Index2`) selects

an integer from among the five positions of the selected array. The syntax of multidimensional arrays may be abbreviated in Pascal. The abbreviations for the example above would be:

```
TYPE
  Matrix = ARRAY [1 .. 3, 1 .. 5] OF INTEGER
VAR
  MArr: Matrix;
  Index1: 1 .. 3;
  Index2: 1 .. 5;
BEGIN
  ...  MArr[Index1,Index2] ...
END
```

The components of MArr can be thought of as being arranged in a two-dimensional table:

MArr[1][1]	MArr[1][2]	MArr[1][3]	MArr[1][4]	MArr[1][5]
MArr[2][1]	MArr[2][2]	MArr[2][3]	MArr[2][4]	MArr[2][5]
MArr[3][1]	MArr[3][2]	MArr[3][3]	MArr[3][4]	MArr[3][5]

where the first subscript indicates a row and the second subscript a column. Thus the number of subscripts is often referred to as the *dimension* of an array. Arrays can have any number of dimensions. For example, a three-dimensional array with

```
TYPE
  ThreeD = ARRAY [0..3, 0..3, 0..3] OF Chip
```

could be used to represent a three-dimensional tic-tac-toe board of 4 planes of 4 × 4 boards.

Multidimensional arrays are often used to represent matrices of numerical values. Suppose that three matrices M1, M2, and M3 are declared as follows:

```
CONST
  Sub1 = 1;
  Sub2 = 3;
  Sub3 = 2;
TYPE
  Matrix1 = ARRAY[1..Sub1, 1..Sub2] OF INTEGER;
  Matrix2 = ARRAY[1..Sub2, 1..Sub3] OF INTEGER;
  Matrix3 = ARRAY[1..Sub1, 1..Sub3] OF INTEGER;
VAR
  M1: Matrix1;
  M2: Matrix2;
  M3: Matrix3;
```

The *product* of M1 and M2 is defined to be a matrix whose type is Matrix3 and whose element [I,K] has the value

$$\sum_{J=1}^{Sub2} (M1[I,J] * M2[J,K])$$

for all I in the range 1..Sub1 and all K in the range 1..Sub3. The second dimension of M1 must be the same as the first dimension of M2 for the product to be defined. Suppose that M3 is to be computed as the product of M1 and M2. Element M3[I, K] is formed by multiplying the corresponding elements of row I of M1 by those of column K of M2. If M1 and M2 have the values shown below:

$$M1=\begin{bmatrix}1 & 2 & 3\end{bmatrix} \quad M2=\begin{bmatrix}4 & 5\\ 6 & 7\\ 8 & 9\end{bmatrix}$$

then the product is:

$$M3=\begin{bmatrix}40 & 46\end{bmatrix}$$

since

```
M3[1,1] = 1*4 + 2*6 + 3*8 = 40
M3[1,2] = 1*5 + 2*7 + 3*9 = 46
```

The following program multiplies two matrices whose values are read from the input:

```
PROGRAM MatrixMultiply(INPUT, OUTPUT);
  {Include Matrix declarations}
  PROCEDURE Mult(VAR In1: Matrix1; VAR In2: Matrix2;
                 VAR Out: Matrix3);
    {For all I,K (1<=I<=Sub1) AND (1<=K<=Sub3)
        Out[I,K] := Sum(J=1..Sub2) (In1[I,J]*In2[J,K])}
    VAR
      I: 1..Sub1;
      J: 1..Sub2;
      K: 1..Sub3;
      Value: INTEGER;
    BEGIN {Mult}
      FOR I := 1 TO Sub1
      DO
        FOR K := 1 TO Sub3
        DO
          BEGIN
            Value := 0;
            FOR J := 1 TO Sub2
            DO
              Value := Value + In1[I,J] * In2[J,K];
            Out[I,K] := Value
          END
    END; {Mult}
  BEGIN {MatrixMultiply}
    {Fill M1}
    FOR I := 1 TO Sub1
    DO
      FOR J := 1 TO Sub2
      DO
        READ(M1[I,J]);
    {Fill M2}
    FOR J := 1 TO Sub2
    DO
      FOR K := 1 TO Sub3
      DO
        READ(M2[J,K]);
    Mult(M1,M2,M3); {M3 := M1 * M2}
    FOR I := 1 TO Sub1
    DO
      BEGIN
        FOR K := 1 TO Sub3
        DO
          WRITE(M3[I,K]);
        WRITELN
      END;
  END {MatrixMultiply}.
```

Execution

```
INPUT :1 2 3 4 5 6 7 8 9
OUTPUT:          40          46
```

19.4.3 Exercises

19.4.1 In many computers the basic unit of data is larger than a character, so several characters can be placed within one unit. The machine operations mostly move whole units, but special operations are available for extracting and replacing part of a unit. For such a machine, PACKED arrays of CHAR are stored with several characters in each unit, and unpacked arrays with one character in each unit. In terms of this implementation, explain each of the following:

a) why more data can be stored in the machine if packed arrays are used than if unpacked arrays are used.

b) why moving a single character from one position to another in a packed array takes more time than if the array were unpacked.

c) why the PACK operation always uses the entire packed array, but only part of the unpacked array may be used.

d) why packed-array elements may not be bound to variable parameters.

19.4.2 It is difficult in Pascal to write a procedure declaration that will handle as actual parameters constant strings. As an example, one might like to pass:

```
PROC('HELLO');
PROC('GOODBYE')
```

but no such procedure PROC can be written.

a) Explain why.

b) Describe the best way around the problem that Pascal allows.

c) Does this difficulty come up with variables as well as constants? Explain.

19.4.3 Modify the program Index (19.4.1) so that each printed word is followed by a line number, of the line on which it *first* appeared. Number lines from 1 as they are input.

19.4.4 Write a module for *sparse table,* a collection of character strings positioned in rows and columns, where most positions in the table are blank. Declare a TYPE Table in which all the strings are stored end-to-end in a CHAR vector in arrival order, and a list of INTEGER values locates the starting and ending positions of each string, and its position in the table. This locator list should be kept in an order so that the table can be printed by going through it sequentially.

Each Table will require a maximum of 500 characters total string storage. The following types and operations are required:

```
CONST
  SMax = 500;
TYPE
  String = RECORD
             Data: PACKED ARRAY[1..SMax] OF CHAR;
                   {The characters of the string in order}
             EndPoint: 0..SMax {The subscript of the
                      final character; 0 for empty string}
           END;
  Loc = RECORD
          Row: INTEGER;
          Col: INTEGER
        END; {Row and Column values begin at 1}
```

```
PROCEDURE ClearTable(VAR AllNew: Table);
  {This routine must be called first for each table
   to be used.  It can be used to initialize the data.}
FUNCTION TableEntryOK(It: String; Pos: Loc;
                      VAR Tab: Table): BOOLEAN;
  {Enter It into Tab at Pos, returning TRUE if no entry
   was there, FALSE if the entry there was overlaid.}
FUNCTION PrintTable(Tab: Table): BOOLEAN;
  {Print the existing table without disturbing it.
   Use as many columns and rows as needed.
   Make the columns equal-sized in an 60-space width.
   Left justify the entries, and where entries are
   adjacent, there must be at least one space between
   them.  If entries would collide or go outside the
   60 spaces, truncate them as necessary, and in
   this case return FALSE.  Otherwise, TRUE.}
```

Write an application of this module in which strings are input with their row and column subscripts, placed in the table, then printed. For this application, the input data (in order: row, column, string):

```
3 2 Corner
1 3 #
5 1 Jumped Over
4 1 This string is not too long to fit
5 3 The Quick Brown Fox Jumped Over the Lazy Dog
7 2 THE END
```

should print as:

```
                          #

                       Corner
This string is not too long to fit properly
Jumped Over                                The Quick Brown Fox

                       THE END
```

19.4.5 Matrix addition is defined component by component. That is, two matrices can be added only if they are the same size, and each component of the sum (which is also of this size) is the sum of the corresponding components of the two. Write a procedure that adds two matrices.

19.5 Analyzing Programs with Arrays

Preview

Most program analysis is made more difficult by the presence of arrays. Subscripts that may be calculated make each array access a potential reference to any part of the whole array.

Array programs in which the subscript values are fixed can be analyzed using each array element as a variable. This is just the approach used for RECORD types, in which the variables named using the field selectors are treated like any other variable. But array elements are unlike fields of records in that subscripts need *not* be constant, and when a subscript value is calculated in a program, the element to which it refers may range over the whole array. The program meaning may well be different for different subscript values, and the only known way to perform the analysis is to treat the whole array as a variable, with its composite value.

Array values are relations of finite domain, that is, ordered pairs in which the first member of each pair is one of the possible subscript values. Assignment of array elements is described in terms of the entire array by introducing a special relation. Let `Arr` be an array and `Sub` a subscript, `Exp` an element value. Then the effect of the assignment:

```
Arr[Sub] := Exp
```

is to alter the old relation value for `Arr` to a new one symbolized as

```
(Arr; Sub:Exp)
```

That is, `(Arr; Sub:Exp)` represents an array value that is the same as `Arr` but with one element changed. This notation allows concurrent assignments to be written for entire array values. For example, the statement

```
A[S] := Ch
```

has the concurrent-assignment meaning:

```
A := (A; S:Ch)
```

The definition of array-element assignment (Section 19.1.2) shows that the array value `(A; S:Ch)` is the same as that of `A` except in the pair(s) with first member `S`, now paired with `Ch`. Although this notation is not part of Pascal, it is convenient to treat it as if it were an array name, and speak (say) of the value of

```
(Arr; Sub:Exp)[I]
```

in a state s. This value is either the value of `Arr[I]` in s, if `I` and `Sub` do not have the same value in s, or it is the value of `Exp` in s, if `I` and `Sub` are the same in s. This may be written

$$\boxed{\texttt{(Arr; Sub:Exp)[I]}} =$$
$$\{<s, \boxed{\texttt{Arr[I]}}(s)>: \boxed{\texttt{I}}(s) \neq \boxed{\texttt{Sub}}(s)\} \cup \{<s, \boxed{\texttt{Exp}}(s)>: \boxed{\texttt{I}}(s) = \boxed{\texttt{Sub}}(s)\}.$$

This formula expresses the fundamental information required for analysis of array programs.

19.5.1 Exchange

The following fragment was used in `ExchangeSort` to switch the values of two elements in the array `Vals`:

```
{exchange Vals[Start] and Vals[Min]}
Save := Vals[Start];
Vals[Start] := Vals[Min];
Vals[Min] := Save
```

The trace table for this fragment is shown below:

Statement	Save	Vals
`Save := Vals[Start]`	`Vals[Start]`	
`Vals[Start] := Vals[Min]`		`(Vals; Start:Vals[Min])`
`Vals[Min] := Save`		`((Vals; Start:Vals[Min]); Min:Vals[Start])`

That is, the effect of these three statements is given by the concurrent assignment:

```
Save,Vals := Vals[Start],
             ((Vals; Start:Vals[Min]); Min:Vals[Start])
```

To prove that these statements do what they were designed to do requires showing that the values at subscript positions `Min` and `Start` are switched. But because an array is involved, that is not all that must be shown. The values at all other subscript positions must be unchanged. Without this additional demonstration, the statements could do arbitrary damage to the array and still be considered correct. In terms of the initial state, the statement that the values are switched can be given using the array value from the trace table. The value v_1 of

```
((Vals; Start:Vals[Min]); Min:Vals[Start])[Start]
```

must be the same as the value of `Vals[Min]`, and

```
((Vals; Start:Vals[Min]); Min:Vals[Start])[Min]
```

must have the same value (v_2) as `Vals[Start]`. Consider the first of these. There are two cases. Suppose that `Start` and `Min` have different values. Then using the fundamental array property, v_1 is the value of

```
(Vals; Start:Vals[Min])[Start]
```

and using the array property again, because the subscript values are the same, v_1 is the value of `Vals[Min]` as required. In the second case, `Start` and `Min` are the same. Then v_1 is the value of `Vals[Start]`, which is `Vals[Min]` as required, since the subscripts are the same. A similar analysis establishes that the value v_2 is correct.

Finally, it must be shown that all other array elements are unchanged, that is, for any `J` value within the bounds (but not the value of `Min` or `Start`), the value

```
((Vals; Start:Vals[Min]); Min:Vals[Start])[J]
```

is the same as that of `Vals[J]`. Applying the array property twice gives

```
(Vals; Start:Vals[Min])[J]
```

and then `Vals[J]` as required.

It is instructive to compare the effort required to verify an exchange between two array elements and two scalar variables, as done in Chapter 6. The proof for array-element exchange is much more difficult, indicating that arrays should be treated with respect, and avoided where possible.

19.5.2 A WHILE Statement Example

Suppose we want to verify that the WHILE-statement:

```
WHILE I <= Size
DO
  BEGIN
    Arr[I] := I;
    I := I + 1
  END
```

computes the function f:

```
(I≤Size →
   I,Arr := Size+1,((...((Arr;I:I);I+1:I+1)...):Size;Size)) |
(I>Size → )
```

Since the WHILE statement terminates for any values of `I` and `Size`, and f is defined for all such values, the domains agree, as required by the first condition of the WHILE verification rule. For any state s such that `I <= Size`(s) is FALSE, the definition gives $f(s) = s$, as required by condition 2 of the rule. It remains only to demonstrate condition 3:

`IF I<=Size THEN BEGIN Arr[I]:=I; I:=I+1 END` $\circ f = f$.

The following execution tables are helpful:

Statement	Condition	I	Arr
`IF ...`	`I≤Size`	`I+1`	`(Arr; I:I)`
f	`I+1≤Size`	`Size+1`	`((...((Arr;I:I);I+1:I+1)...);Size:Size)`

The final line in this table is tricky to calculate, since it involves replacing `I` by `I+1` and `Arr` by `(Arr;I:I)` in:

```
((...((Arr;I:I);I+1:I+1)...);Size:Size)
```

Since the new value of `Arr` depends on the value of `I`, the latter is substituted first:

```
((...((Arr;I+1:I+1);I+2:I+2)...);Size:Size)
```

then substitute for `Arr`:

```
((...(((Arr;I:I);I+1:I+1);I+2:I+2)...);Size:Size)
```

as the table shows. In this case the the condition is

```
(I<=Size) AND (I+1<=Size)
```

which is the same as `I<Size`. The assignment is:

```
I, Arr := Size+1, ((...((Arr;I:I);I+1:I+1)...);Size:Size)
```

The next table:

Statement	Condition	I	Arr
IF ...	I≤Size	I+1	(Arr; I:I)
f	I+1>Size		

results in the condition `I=Size` and the assignment

```
I, Arr := I+1, (Arr; I:I)
```

and in this assignment `Size` may replace `I` since they are the same in this case.

The next table contributes nothing to the meaning of the composition being calculated, because its condition is contradictory:

Statement	Condition	I	Arr
IF ...	I>Size		
f	I<=Size	Size+1	((...((Arr;I:I);I+1:I+1)...);Size:Size)

The final table has the condition `I>Size` and an identity assignment:

Statement	Condition	I	Arr
IF ...3	I>Size3	3	3
f	I>Size		

The combined results are identical to *f*, with the first case broken into two:

```
(I<Size →
   I,Arr := Size+1,((...((Arr;I:I);I+1:I+1)...);Size:Size) ) |
(I=Size → I,Arr := Size+1, (Arr; Size:Size) ) |
(I>Size → )
```

Although the meaning of the WHILE statement has been demonstrated to be *f*, the form of the final array value:

```
((...((Arr;I:I);I+1:I+1)...);Size:Size)
```

does not give much insight into the values of the individual array elements. What is needed is to calculate the value

```
((...((Arr;I:I);I+1:I+1)...);Size:Size)[J]
```

for an arbitrary value of `J` that lies between the values of `I` and `Size`. Applying the fundamental array property until the term with subscript `J` is reached, the value is `J`. Thus the WHILE statement has set the array value to an identity mapping in which each subscript contains its own value.

Again it is evident that verifications with arrays require much effort, corresponding to the flexibility and power they provide for optimizing execution. When an array is used in a controlled way, for example as part of the implementation of a module, then the power may be worth the cost, because the complications of analysis are confined to the proof that the module is correct. But

uncontrolled use of arrays is to be avoided.

19.5.3 Exercises

19.5.1 Simplify the value:

```
(((X;I:X[I-1]);J:(X;K:Y)[K]);I-1:Z)[I]
```

using the fundamental property of arrays.

19.5.2 For the fragment that exchanges two values from array `Vals` (Section 19.5.1), try to construct a trace table using

```
Vals[1]    Vals[2]  ...  Vals[10]
```

as the column headings. Explain what goes wrong.

19.5.3 Suppose a programmer means to set one array element to zero, and writes

```
A[I] := 0
```

In subsequent debugging it is discovered that the subscript should be `I+2`, but the first statement is erroneously left in the program:

```
A[I] := 0;
A[I+2] := 0
```

The programmer now wishes to prove that the program is correct.

a) In terms of the complete value for array `A`, write the statement that subscript position `I+2` has been zeroed, and prove that it is so.

b) Explain why the proof of a) is insufficient.

c) Give a statement of what *should* be proved, and show that it does *not* hold for the the erroneous program.

19.5.4 Show that the Pascal fragment

```
A[K] := A[K] + K;
K := K + 1;
A[K] := A[K] - K;
K := K - 1
```

computes the function

```
A[K], A[K+1] := A[K]+K, A[K+1]-K-1
```

(Don't forget to show that the other array elements are unchanged by the fragment.)

19.5.5 Show that the Pascal fragment

```
K := 0;
X[0] := 1;
WHILE K <= N
DO
  BEGIN
    K := K + 1;
    X[K] := X[K-1] * K
  END
```

computes the function

```
For all I 0<=I<=N, X[I],K := I!,N
```

(Hint: do the analysis using the whole array value, and go to the given form only at the end.)

19.5.6 Show that the Pascal fragment

```
K := A;
WHILE K <= B
DO
  BEGIN
    K := K + 1;
    X[K-1] := X[K]
  END
```

computes the function

```
For all I A<=I<B, X[I],K := XO[I+1],B
```

where XO stands for the initial value of the array X. (Hint: express the function in terms of the whole array value.)

19.6 Chapter Summary

Arrays allow efficient implementations of finite versions of abstract data types based on lists such as files, stacks, and queues. They also permit efficient implementations of functions, particularly functions for which no simple computational rule is available. In fact the subscript notation is suggestive of that for function values.

However, there is another side to this power. Used in an undisciplined way, arrays represent design liabilities because of their generality and power. The data types of D Pascal provide not only disciplined access to data in a list, but also the intellectual tools to help assure success in program design. For example, the use of a file can ensure that every record of interest has been examined, whereas a solution employing random access to the same set of records in an array can skip some inadvertently. As a rule, it is easier to improvise solutions with arrays than with more disciplined, list-based data types, but much more difficult to achieve correct solutions.

These differences lead to a program design strategy that is fundamental for the rest of this book. In order to get the best of both random access and disciplined access data types, it is a good idea to observe the following guidelines:

Use arrays primarily for the construction of abstract data types which access data in disciplined ways, but in ways best suited to the solution being implemented.

Never use arrays in ways that jeopardize your practical ability to carry out correctness arguments about your programs.

Recognize, in early design thinking about any program, that arrays are seldom suitable for high-level design. That is, design in D Pascal and optimize in O Pascal.

Design for function first and performance second. Get your solution right with a preponderance of disciplined data types, and optimize in a stepwise reorganization of your solution.

CHAPTER 20

LINKED STRUCTURES

Chapter Preview

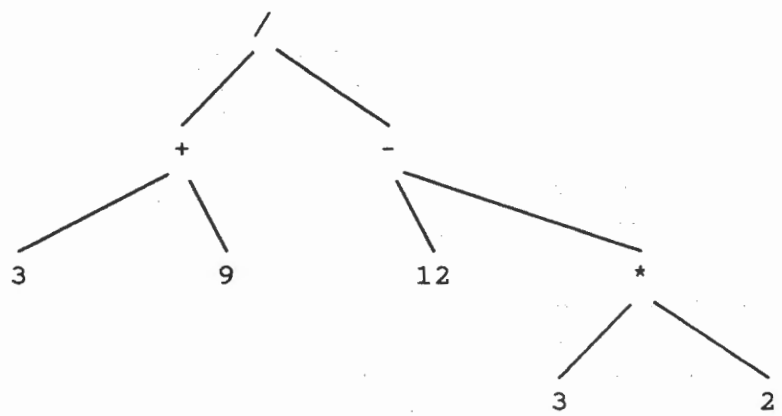
Many important data structures grow and change as they are used by a program during execution. These linked structures can be kept in Pascal array storage, but they are more naturally implemented using Pascal pointer types.

All Pascal data types except files are static: the size of stored data is fixed by the declaration. Ordinal and set data types have trivial data structures of a single value. Records and arrays comprise multiple values, but the structure and number of components is fixed. Files are of one generic structure, the sequence, and can vary dynamically only in length.

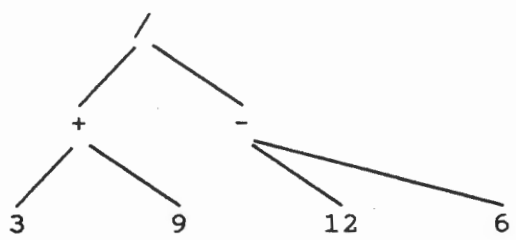
However, many problems are best solved with data structures that vary in size and shape as the solution progresses. For example, an arithmetic expression such as:

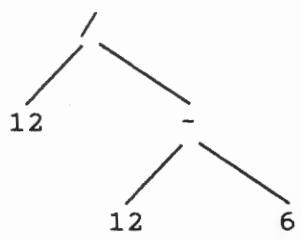
```
(3+9)/(12-(3*2))
```

can be represented as the tree:

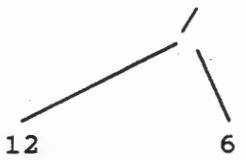


The evaluation of the expression, by repeatedly replacing innermost parenthesized expressions by their values, can be viewed as tree pruning, replacing a lowest subtree representing a binary operation by the result of that operation. In this process the tree is transformed successively into new trees until only one node remains, containing the result of evaluation.





(First the multiplication, then the addition has been performed.)



(The subtraction, then the division, yield the final result 2.)

Each tree in the sequence can be viewed as an instance of a single data structure varying dynamically during the solution.

20.1 Implementing Linked Structures With Arrays

Preview

An array of records can be used to hold a linked structure by adding a "pointer" field to the record, to record the subscript values of other records in the structure.

Static array structures can be viewed as dynamic if their components move relative to each other. For example, suppose a sequence of records is stored in an array, and a sorted version of the data is required. `ExchangeSort` or `MergeSort` (Section 19.3.2) can be used to rearrange the records, moving the entire record based on all or part of its contents. (The portion of a record that actually participates in the ordering is called the *sort key*.) But a better way to sort the data is to add a new field to each record and place in it the subscript of the alphabetically following record. The records are chained together by these subscript pointers, and by following the chain, they can be obtained in sorted order. The advantage over physically sorting the records is that only the chain pointers need be changed. When the records are large, particularly when the sort key is a small part of a large record, chaining is much faster than moving the records.

20.1.1 A Simple Linked List

The array `PRecs` might be designed to contain personnel information for a group of people:

```
CONST
  NameLen = 7;
  AddrLen = 25;
  Max = 4;
TYPE
  Month = (Jan, Feb, Mar, Apr, May, June,
           July, Aug, Sept, Oct, Nov, Dec);
  Sex = (Female, Male);
  Date = RECORD
           Mo: Month;
           Day: 1..31;
           Year: INTEGER
         END;
  Person = RECORD
             Name: PACKED ARRAY[1..NameLen] OF CHAR;
             SocialSecurity: INTEGER;
             Address: PACKED ARRAY[1..AddrLen] OF CHAR;
             BirthDate: Date
           END;
VAR
  PRecs: ARRAY [1..Max] OF Person;
```

Suppose that the following data are stored in this structure:

Index	PRecs[Index].Name	rest of record
1	**Miller**□	...
2	**Smith**□□	...
3	**Plane**□□	...
4	**Jones**□□	...

Consider the Name field of each record as the key. The most straightforward way to get a sorted version of the data is to move the records about, to eventually produce the array:

Index	PRecs[Index].Name	rest of record
1	**Jones**□□	...
2	**Miller**□	...
3	**Plane**□□	...
4	**Smith**□□	...

The second way to get a sorted version is to add a 0..Max field Next to each record and declare a new variable of type 0..Max, say First. Then, leaving the records in place, set the Next field of each record to the subscript of the one following in alphabetical order, and First to the subscript of the initial record in that order (4):

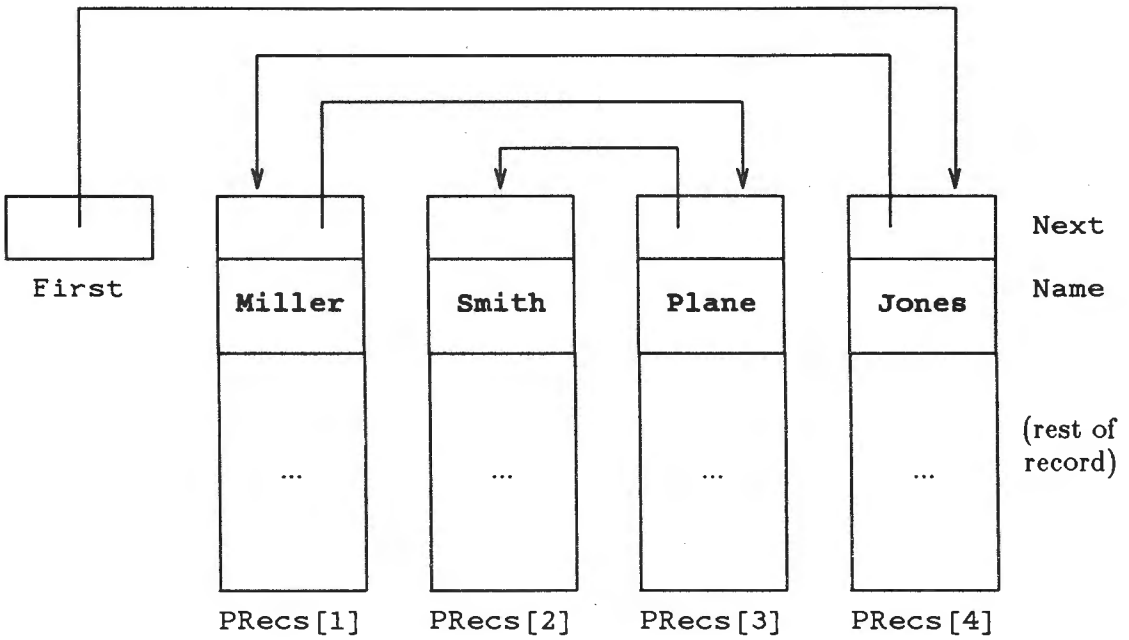
Index	PRecs[Index].Next	PRecs[Index].Name	rest of record
1	3	**Miller**□	...
2	0	**Smith**□□	...
3	2	**Plane**□□	...
4	1	**Jones**□□	...

The last record is flagged with the sentinel 0, an impossible subscript for this array indicating that nothing follows. Now to obtain the record subscripts in alphabetical order of their contents, begin with First and follow the chain:

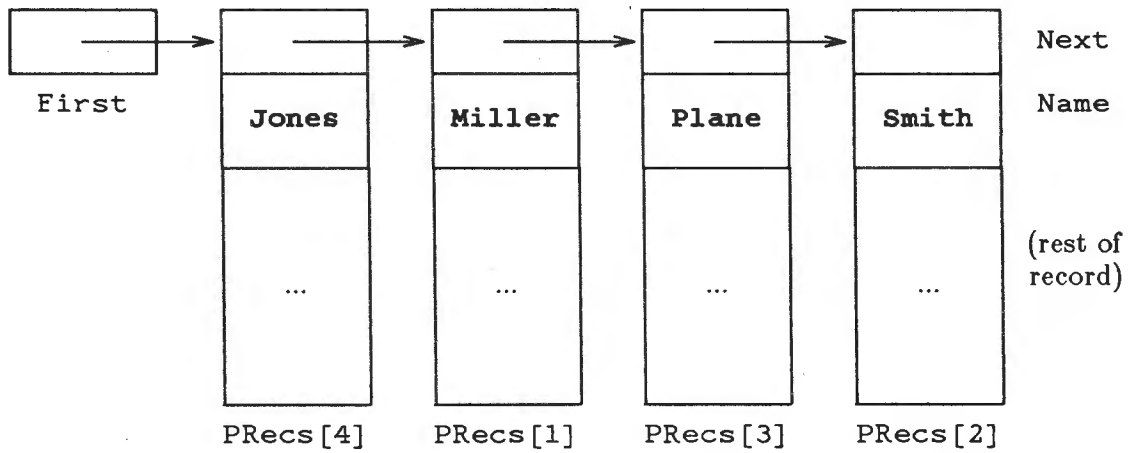
```
Index := First;
WHILE Index <> 0
DO
  BEGIN
    {Index is the subscript of the current
     item in alphabetical sequence.  Use
     it to access whatever fields are needed.}
    Index := PRecs[Index].Next
  END
```

The series of values taken on by Index inside the WHILE statement is thus 4,1,3,2—the sorted order.

The chain pointers can be drawn in to give a graphic representation of the linked structure. The subscript value in each record is replaced with an arrow pointing to the record with that subscript:

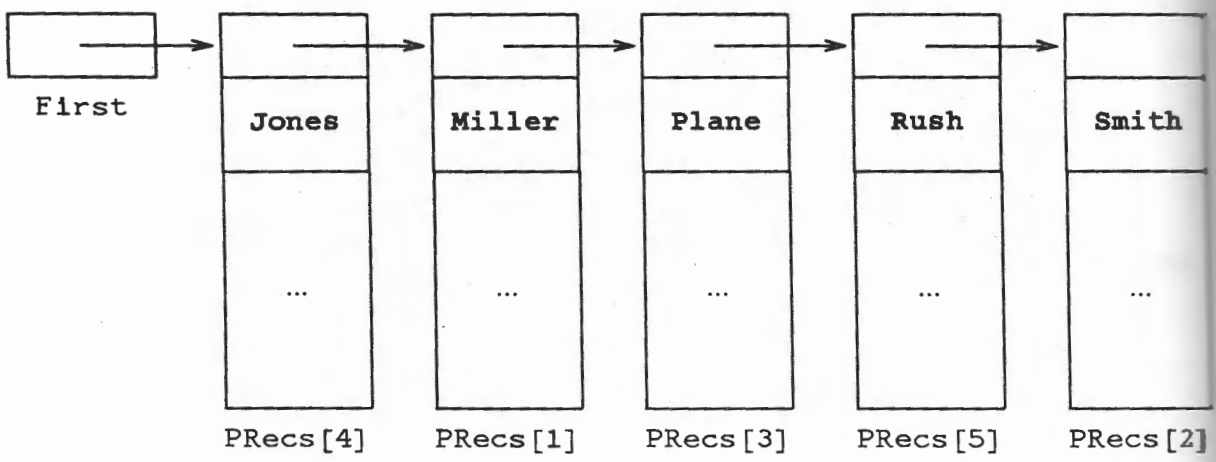


However, a simpler diagram results if the array sequence is abandoned in favor of the sort sequence.



A structure in which records contain pointers to other records is called a *linked list*. The links, stored in each record's `Next` component, provide a sort sequence by specifying, for each record, the location of its alphabetic successor. When there is no link (indicated here by a 0 value), the record is the last in the linked list.

Records may be added to a linked list without moving even the links for most of the existing records. For example, if a new record with the `Name` **Rush**□□□ should be added to the alphabetic sequence, the array can be expanded to place it without regard for the sequence, and the links rearranged.



The linking requires just two assignments no matter how large the list may be:

```
PRecs[5].Next := PRecs[3].Next;
PRecs[3].Next := 5
```

Similarly, any record in the list may be deleted with a single assignment.

20.1.2 Insertion Sort

A linked list is the ideal data structure for creating a sorted list of large records as they are read one by one. Beginning with an array large enough to contain all the records expected, the records can be placed in the array in sequential order, and links created to maintain sorted access as each record is added. In the example below the records consist of just a single-character key and all the input characters are on a single line. Since the design does not depend on the form of input record, it can be used as a model for sorting records of any type, modifying only the TYPE declaration.

Design Part 1

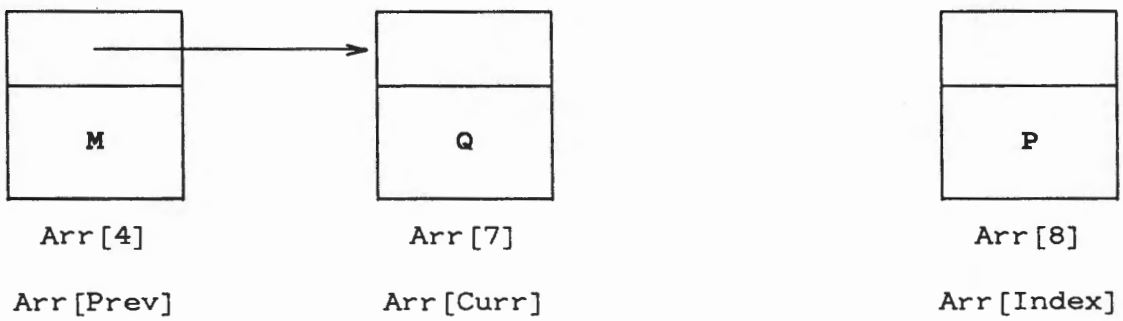
```
PROGRAM InsertionSort(INPUT,OUTPUT);
{sort the characters in INPUT into ascending order}
  CONST
    Max = 16;
    ListEnd = 0;
  TYPE
    RecArray = ARRAY[1..Max] OF
                 RECORD
                   Next: 0..Max;
                   Key: CHAR
                 END;
  VAR
    Arr: RecArray;
    First, Index: 0..Max;
    Extra: CHAR;
```

```
BEGIN {InsertionSort}
  First := 0;
  Index := 0;
  {Place record in linked list if space permits,
   otherwise ignore with error message}
  WHILE NOT EOLN
  DO
    BEGIN
      Index := Index + 1;
      IF Index > Max
      THEN
        BEGIN
          READ(Extra);
          WRITELN('Record containing ',
            Extra, ' ignored.');
        END
      ELSE
        BEGIN
          READ(Arr[Index].Key);
          {link Arr[Index] into existing list}
        END
    END;
  {print list starting with Arr[First].Key}
END.  {InsertionSort}
```

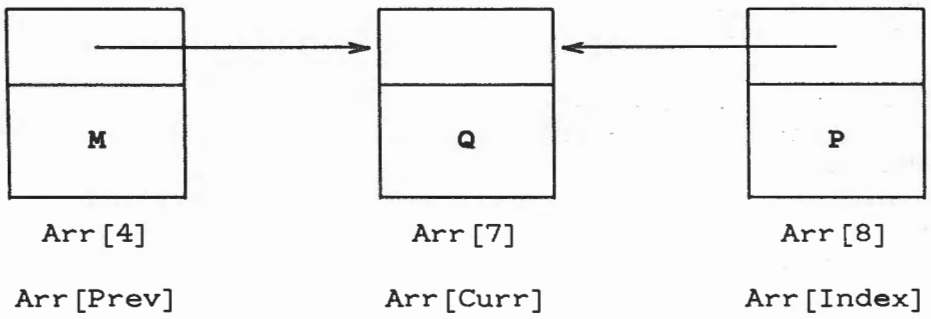
To find the point at which to insert the newly obtained key `Arr[Index].Key`, a new variable `Curr` is assigned successive values of the link field until:

```
Arr[Index].Key <= Arr[Curr].Key
```

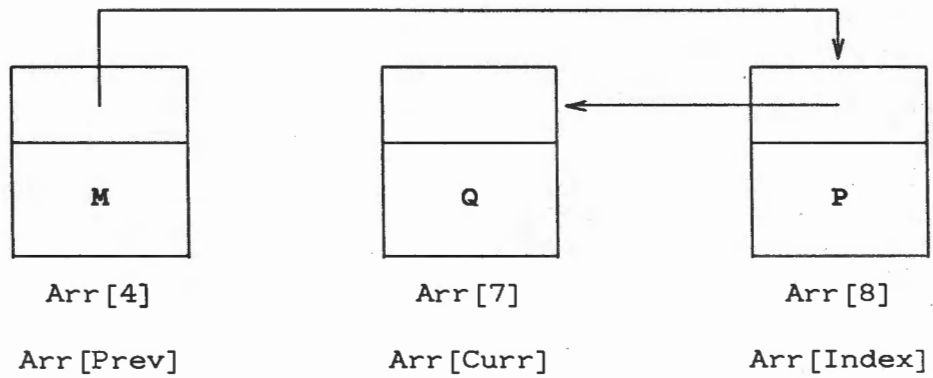
is TRUE or the end of the list is reached (`Curr` takes value 0). Assigning `Arr[Index].Next` the value of `Curr` links the new value to the values that should follow it in the list, but does not link the beginning of the list to the new value. Another variable `Prev` can be made to point at the element before that of `Curr`, which allows the second part of linking to be completed. For example, consider the situation in which a new item has been read into `Arr[8]` (`Index` is 8), and the item should be inserted between subscripts 4 (`Prev`) and 7 (`Curr`). Before the insertion, this portion of the linked list appears as:



Assigning `Arr[Index].Next` the value of `Curr` (that is, linking the end of the list to the new record) results in:



Assigning `Arr[Prev].Next` the value of `Index` links the beginning of the list to the new record:



Some care must be taken when linking the first part of the list to the new value since this value may become the new first element of the list. If `Prev` begins with value 0 and does not change during the search, the new element is at the beginning of the list, and must be pointed to by `First` rather than by another list element. No similar caution is required should the new element fall at the end of the list, because the 0 value in the old final record would be transferred to the new record as if it were an onward pointer.

Design Part 1.1

```
{link Arr[Index] into existing list}
Prev := 0;
Curr := First;
{find values for Prev and Curr (if they exist)
 such that
 Arr[Prev].Key <= Arr[Index].Key <= Arr[Curr].Key};
Arr[Index].Next := Curr;
IF Prev = 0
THEN
  First := Index
ELSE
  Arr[Prev].Next := Index
```

A Boolean variable `Found` is introduced to stop the search as soon as `Prev` and `Curr` have acquired their values.

Design Part 1.1.1

```
{find values for Prev and Curr (if they exist)
 such that
 Arr[Prev].Key <= Arr[Index].Key <= Arr[Curr].Key}
Found := FALSE;
WHILE (Curr <> 0) AND NOT Found
DO
  IF Arr[Index].Key > Arr[Curr].Key
  THEN
    BEGIN
      Prev := Curr;
      Curr := Arr[Curr].Next
    END
  ELSE
    Found := TRUE;
```

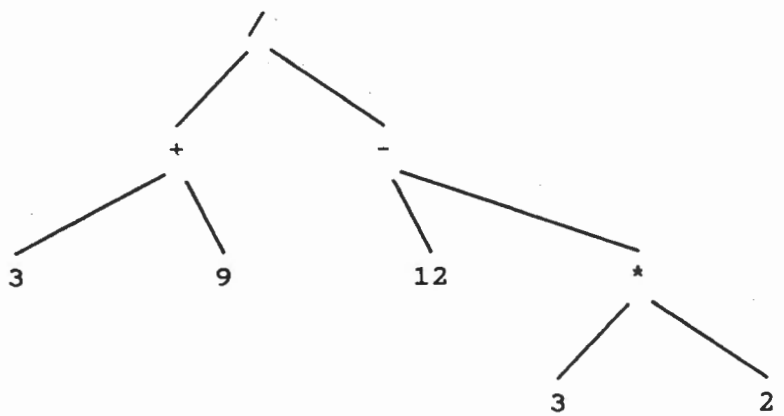
The final design part prints the values in sorted order.

Design Part 1.2

```
{print list starting with Arr[First].Key}
Index := First;
WHILE Index <> ListEnd
DO
  BEGIN
    WRITE(Arr[Index].Key);
    Index := Arr[Index].Next
  END;
WRITELN
```

20.1.3 Evaluating Arithmetic Expressions

An expression tree like:



can be stored as an array of records of the form:

```
CONST
  ESize = 9;
TYPE
  Node = RECORD
           Left, Right: 0..ESize;
           Op: CHAR;
           Val: INTEGER
         END;
VAR
  Exp: ARRAY [1..ESize] OF Node;
```

where `ESize` is the maximum number of nodes in the expression tree. Each record (node) can contain two pointers to the information below it in the tree (`Left`, `Right`), a character for the arithmetic operation (`Op`), and an integer operand (`Val`). A particular node will contain either an operation or an operand, but not both. Operation nodes will have pointers, but operand nodes will not. In the structure, the expression above appears as:

Node	1	2	3	4	5	6	7	8	9
Left	2	4	6	0	0	0	8	0	0
Right	3	5	7	0	0	0	9	0	0
Op3	/	+	-				*		
Val				3	9	12		3	2

where node N is stored in position N of the array `Exp`. An operation can be performed if an operator node is linked to two operand nodes. The nodes numbered 2 and 7 have this property. When an operation is performed, the resulting value can replace the operation node. This is accomplished by erasing the node fields corresponding to the `Left` and `Right` links and the operation, and assigning the result of the operation to the `Val` field. For example, if the operation at node 2 is carried out, the representation of the expression would change to:

Node	1	2	3	4	5	6	7	8	9
Left	2	0	6	0	0	0	8	0	0
Right	3	0	7	0	0	0	9	0	0
Op3	/		-				*		
Val		12		3	9	12		3	2

If the operation at node 7 is carried out as well, the resulting data structure is:

Node	1	2	3	4	5	6	7	8	9
Left	2	0	6	0	0	0	0	0	0
Right	3	0	7	0	0	0	0	0	0
Op3	/		-						
Val		12		3	9	12	6	3	2

Now the operation at node 3 can be carried out:

Node	1	2	3	4	5	6	7	8	9
Left	2	0	0	0	0	0	0	0	0
Right	3	0	0	0	0	0	0	0	0
Op3	/								
Val		12	6	3	9	12	6	3	2

Finally the completed evaluation is obtained by performing the node-1 operation:

Node	1	2	3	4	5	6	7	8	9
Left	0	0	0	0	0	0	0	0	0
Right	0	0	0	0	0	0	0	0	0
Op									
Val	2	12	6	3	9	12	6	3	2

A program could be designed for evaluating arithmetic expressions stored in the structure above. To recognize operator nodes, a character (say `!`) could be placed in the `Op` field of each *operand* node. The recognition of operator nodes that can be immediately evaluated can then be accomplished by searching for an `Op` value that is not `!`, but whose `Left` and `Right` links go to `Op` values that are `!`. Some efficient ways to construct and use expression trees are suggested in the Exercises.

20.1.4 Tree Structures

The preceding examples of linked data structures illustrate two fundamental principles:

1. Every node is described by a single data type, which contains at least one link component.
2. The values of the links determine the data structure.

At least one link into a linked data structure exists outside the structure. Sometimes this link is explicit, as `First` is in the list of personnel records; sometimes it is implicit, as in the expression trees, where the root of the tree is stored in the first array position.

While the values of the links determine a data structure at each instant, unless those values correspond to some recognizable pattern such as a list or tree, the semantics of the data structure itself (not of the array that holds it) can become very complex. It is therefore valuable to have standardized linked structures that can be employed as required, and the *k-trees* are such a family. A k-tree, for integer $k > 0$, is a tree in which every interior node or *crotch* has exactly k branches, while every end node or *leaf* has no branches. For example, a 1-tree is a simple linked list like that used for the personnel records, a 2-tree is a binary branching tree like that used to represent arithmetic expressions. The nodes of a k-tree each have k links. Since each node has only one pointer to it, the tree is said to be *singly-linked.*

Recursion is a valuable technique in processing k-trees. For example, suppose that an arithmetic expression is stored in a 2-tree with each crotch an operator node and each leaf an operand. Evaluation can be accomplished as follows:

> If the root is a leaf, the value is the value of the leaf.
>
> Otherwise (the root is a crotch), evaluate the left and right sub 2-trees and compute the value using their values with the operation at the root.

The evaluation is recursive because evaluation of an operation node requires evaluation of the nodes to which it is linked, which may in turn be operation nodes.

When information is stored in a k-tree, it is often necessary to *traverse* the tree in some fashion, following the links in some particular way, for example, to examine each leaf. Navigating around in a k-tree is easy so long as the links are followed, but often it is convenient to go backwards. (For example, when information is found at a leaf, to obtain the node immediately above it.) This can be done in a single-linked tree only by storing extraneous information during the traversal, or by starting again from the root. In a *doubly-linked* k-tree, one link is added to each

node N (making k+1 total links), which points to the node that points to N. Doubly-linked trees permit easier navigation, but are larger and more difficult to construct and modify, since more links must be maintained.

20.1.5 Exercises

20.1.1 In the example of the linked list containing five names at the end of Section 20.1.1:

a) give a single assignment statement that will delete **Miller** from the list.

b) Will an assignment similar to a) do as well for any other record in the list? Explain.

20.1.2 Suppose that names are stored in an array of PACKED arrays, as in PRecs in Section 20.1.1.

a) Modify ExchangeSort (Section 19.3.2) to sort the names.

b) How can a name consisting of the usual first, last, and middle parts be stored in an array of PACKED arrays so that names may be easily sorted and searched in the ways that might be required by a personnel office?

20.1.3

a) Devise a single line of test data for InsertionSort (Section 20.1.2) that will exercise *all* of the following situations:

The inserted item goes at the beginning of an existing list.

The inserted item goes at the end of an existing list, and the following inserted item goes at the end of *that* list.

The inserted item goes somewhere between the beginning and end of the list.

b) This kind of testing is called *special-values* or *boundary-values* testing because it exercises special cases in the data structure. Explain why for InsertionSort, an item falling at the end of the list isn't really a boundary value, but one falling at the beginning is.

20.1.4 Modify InsertionSort (Section 20.1.2) so that after it has constructed the list, it reads a second line of input containing an integer P, and prints record number P in the list. Be sure to handle error cases.

20.1.5 *Program animation* consists of adding graphic capabilities to a program so that as it executes, a pictorial record of its actions are produced. Animation is valuable in understanding how algorithms work, and in debugging programs. Think about animating InsertionSort (Section 20.1.2). What should be printed at which points in execution? Give an example of what you would like the output to look like for a typical addition of one record.

20.1.6 Consider the sample expression tree at the start of Section 20.1.3.

a) What is the value of Exp[Exp[Exp[1].Left].Right].Val?

b) What array expression will access the character * ?

20.1.7 Using an array whose index type is

```
TYPE Color = (Violet, Indigo, Blue, Green, Yellow, Orange, Red)
```

create a data structure that can be used to read in a list of colors and sort them into "rainbow order" (the order of the enumerated type above) using a linked list. The color names are in the input as initial letters on a single line, but the full names should be printed. For example, the response to the input †**YIG**† should be

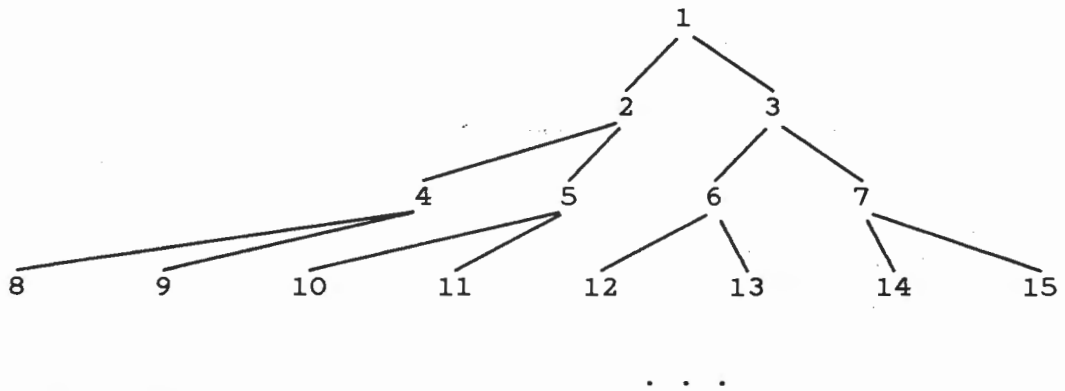
```
Indigo
Green
Yellow
```

20.1.8 Write a program to do expression evaluation by searching the structure of Section 20.1.3 for an operation that can be performed, performing it, and repeating the search.

20.1.9 Replace the expression structure of Section 20.1.3 with a doubly-linked one in which each operation node contains a back pointer to the operation node before it, and write a program to do expression evaluation in this structure by following the back pointer after each operation is completed.

20.1.10 Write a program to do recursive expression evaluation using the structure of Section 20.1.3.

20.1.11 In a 2-tree, all the possible node positions can be numbered starting with the root as 1, as follows:



. . .

If these position numbers are used as the subscripts in which records are stored, no pointers need be stored; instead, the links can be *computed*. For a record stored at position N, the location of its left subtree is position $2N$, and its right subtree is position $2N+1$. Furthermore, the backpointer from node M is to node $M \div 2$.

a) Give the declaration of the suitable 2-tree data structure for arithmetic expressions.

b) Write a recursive evaluation program for the structure of a).

c) Write a nonrecursive *depth-first* search procedure that examines a 2-tree stored as in a) by first going all the way down the left subtree, then back to the first node with an unsearched subtree and down its left subtree, etc.

20.1.12 Polynomials can be represented by linked structures. Declare a type and variable of that type to represent a polynomial in the variable x with integer coefficients. Give the values that represent $x^2 - 2x + 1$.

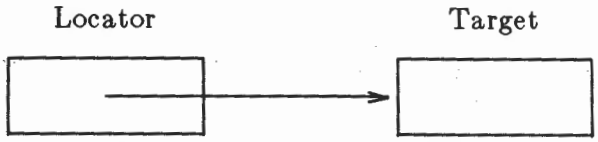
20.2 Pointers

Preview

Pascal pointer objects make precise the idea of linking between parts of a structure. In addition, they allow programs to create and use structures whose size truly varies as the program executes.

Pascal has a collection of data types to support the construction and manipulation of linked structures, called *pointers*. Pointers are unusual data objects, because they have associated with their identifiers *two* values. One value is a Pascal data item of the usual kind, an example of a Pascal TYPE. The other value is something new, a kind of "locator" for the data value. The locator value is really what the pointer is all about. The reason for introducing this new locator between the identifier and the value is that in a linked structure operations are often not on the values stored in the structure, but only on objects that allow the values to be found, later. For example, in building the list of values for an insertion sort, most of the processing involves the links between the values, not the values themselves. In fact, the links are called "pointers" because their values are not the data, but only help to find the data. We will call the "locator" value the *pointer value* (or

sometimes, for clarity, the value of the *pointer itself*). The value pointed at will be called the *target value*. A pointer value for identifier `Ptr` is thus represented by the diagram:



One pointer type is declared by prefixing a type identifier with an upward arrow, "↑". The rules of type syntax are extended as follows:

<new type> ::= <new ordinal type> | <new structured type> |
<new pointer type>

<new pointer type> ::= ↑ <type identifier>

For example:

```
TYPE
  RefInt = ↑INTEGER;
```

`RefInt` is the type of a pointer that locates an integer value. A pointer variable is declared using a pointer type, e.g.:

```
VAR
  PInt: RefInt;
```

The operations for any pointer type are the relationals = and <>, and assignment. When pointers are assigned, it is the value of the pointer itself that gets copied, not the target value. Two pointers are equal if their pointer parts are the same; otherwise, even though they may point to target values that are the same, the pointers are not equal. There is one constant value for a pointer itself: `NIL`. When a pointer variable is assigned this constant, e.g.,

```
PInt := NIL
```

there is no target value to which `PInt` points. Pointers can be given values only by assignment (to `NIL`, or to another pointer with a value), or with the NEW statement, which creates both the object and the locator values. The built-in procedure `NEW` is given a pointer variable as actual parameter. For example, the statement

```
NEW(PInt)
```

creates a new pointer-itself value for `PInt`, which locates (points to) an `INTEGER` target value (as yet unknown). The target of pointer `PInt` is accessed by suffixing `PInt` with an upward arrow:

```
PInt↑
```

This is often called *dereferencing* a pointer value. Thus the target of a pointer has the same form as a file buffer. The rules of syntax are extended as follows to include targets:

<variable access> ::= <variable identifier> | <buffer variable> |
<component variable> | <identified variable>

<identified variable> ::= <pointer variable> ↑

<pointer variable> ::= <variable access>

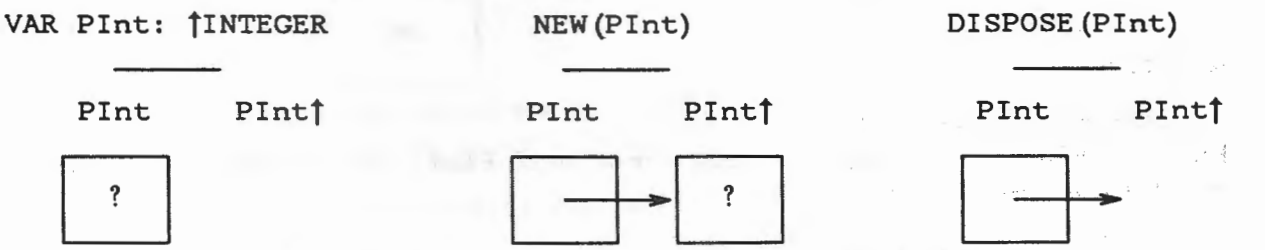
The `DISPOSE` built-in procedure takes as parameter a pointer expression.

```
DISPOSE(PInt)
```

releases the storage for the *target* pointed at by `PInt`. The pointer itself is not released, nor is it necessarily set to `NIL`. Thus, `NEW` and `DISPOSE` are used in pairs separated by the computation that uses a target object:

```
NEW(PInt);
{Make use of the target pointed to by PInt};
DISPOSE(PInt)
```

Pointers and structures built from pointers are best described by pictures. It is conventional to represent pointer variables and targets by boxes, with an arrow between them representing the value of the pointer itself. The diagram below summarizes the sequences of operations above.



Some Pascal machines set the value of a pointer to NIL as a part of the DISPOSE statement, but if this is not done (as in the diagram above), the pointer is said to be a *dangling reference*. It points to a target that has ceased to exist. When a pointer value itself is NIL there is no target, but the programmer can test for this condition:

```
IF PInt <> NIL
 THEN
   {Use PInt↑ ...}
```

However, only careful programming, keeping track of which pointer values are valid, prevents the use of dangling references.

20.2.1 Pointer and Target Values in the Execution State

Pointers are unique in Pascal in allowing the creation of data objects whose size varies as the program executes. The NEW statement creates two objects that did not exist before it was executed: the pointer itself and the target. Since NEW statements can be executed over and over, the number of objects is potentially unlimited. However, there is a naming problem associated with the creation of many objects: they cannot all have identifiers. Any program declares a finite number of identifiers, of which say N are pointer variables. Then N NEW statements can create N pointer values for these variables, and N targets. There is no reason why another NEW statement cannot be executed, but which pointer variable should be used? Suppose one of the N is reused. Then a new pointer value and object are created, *and the former values are lost.* Thus there are still only N pointer values and targets that can be used. The lost values have no connection with any identifier, and so cannot be accessed. The answer to this riddle lies in making a place *within a target object* for a pointer value. Then each time a target is created by a NEW statement, a place is created for another pointer value, and there is no limit to the number of objects the program may create and use. A linked list is a structure of exactly this kind: there is one identifier that points to the beginning of the list, then each record of the list contains a pointer to the next record, but there are no identifiers associated with these records, so a single variable effectively controls an object of arbitrary size.

The existence of pointer values without identifiers poses a problem in showing them within an execution state. Names are needed for the pointer values when no pointer variable is used, and we take these names from the collection {↑1, ↑2, ↑3, ...}, that is, positive integers preceded by an up arrow. In any given state some of these values may already appear; when a NEW statement creates another pointer value, the unused name with the smallest integer is selected. The NEW statement creates not only (say) ↑3 as a pointer value itself, but a target, which we write with the arrow on the other side: 3↑. For example, suppose that beginning with a state containing no pointer values, say

{Ch·X}

(containing one CHAR identifier and its value), the declaration

```
VAR
  PInt: ↑INTEGER
```

is encountered. The state then becomes

{Ch·'X', PInt·?}

in the usual way. Following the statement

```
NEW(PInt)
```

it becomes

{Ch·'X', PInt·↑1, 1↑·?}

because the first pointer name is used for the value of `PInt` itself, and the target to which it points is as yet unknown. Finally,

```
PInt↑ := 99
```

yields

{Ch·'X', PInt·↑1, 1↑·99}.

In simple cases like this, `PInt↑`·99 might just as well have been put in the state instead of 1↑·99, and `PInt` left out altogether. But two complications make our notation essential: pointers may be duplicated, and there may be no identifier for the target value.

First, consider a duplicate pointer. If the declaration is

```
PInt, QInt: ↑INTEGER
```

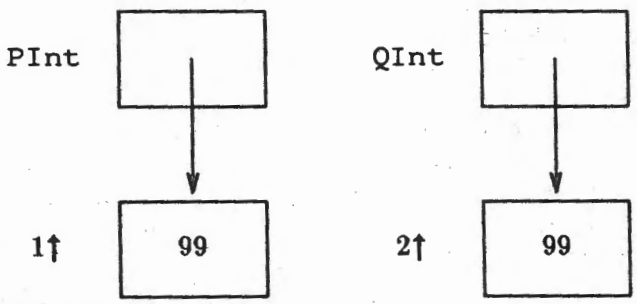
then the code

```
NEW(PInt);
NEW(QInt);
PInt↑ := 99;
QInt↑ := PInt↑
```

leads to the state

{PInt·↑1, QInt·↑2, 1↑·99, 2↑·99},

which has the diagram:



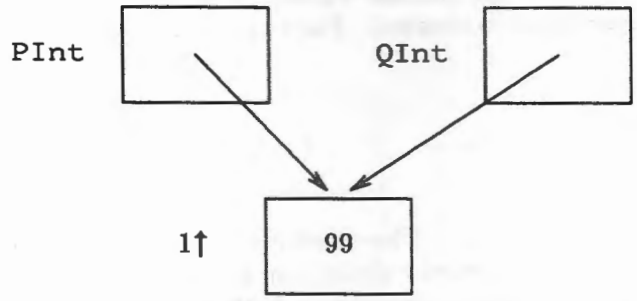
But with the same declarations,

```
NEW(PInt);
PInt↑ := 99;
QInt := PInt
```

yields the rather different state

{PInt·↑1, QInt·↑1, 1↑·99},

with diagram:



In the second case, the pointer values themselves, that is, the values of `PInt` and `QInt`, are equal, and so are the object values (`PInt↑` and `QInt↑`); in the first case only the target values are equal, and the notation shows this.

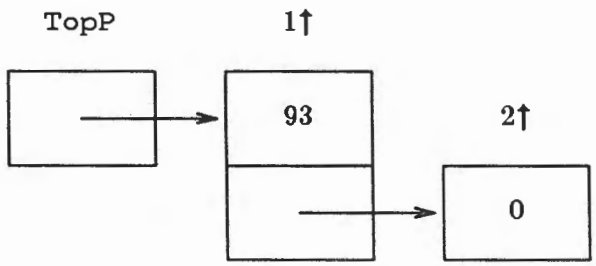
Second, consider a target value without an identifier.

```
TYPE
  ContP = RECORD
            Int: INTEGER;
            PInt: ↑INTEGER
          END;
VAR
  TopP: ↑ContP;
BEGIN
  NEW(TopP);
  TopP↑.Int := 93;
  NEW(TopP↑.PInt);
  TopP↑.PInt↑ := 0
END
```

leads to the state

{TopP=↑1, 1↑.Int=93, 1↑.PInt=↑2, 2↑=0}

in which the target 2↑ with value 0 has no declared identifier. The diagram is:



If a linked list is being constructed, this situation is carried to the extreme that no target but the first item in the list has an identifier; the rest have only pointed-at places.

20.2.2 Implementing Linked Structures with Pointers

Pointers are not very useful if they only name a single value as in the type `RefInt` above. Trying to simulate an array of 20 integers by performing that number of NEW statements:

```
FOR Index := 1 TO 20
DO
  NEW(PInt)
```

creates 20 targets, but overlays the pointers to the first 19. There is no way to recover access to these 19 targets, or even to delete them. A target to which no pointer points is called *garbage*. When pointer values are created by NEW they must be saved, if not in the NEW parameter, then by assigning them to other storage.

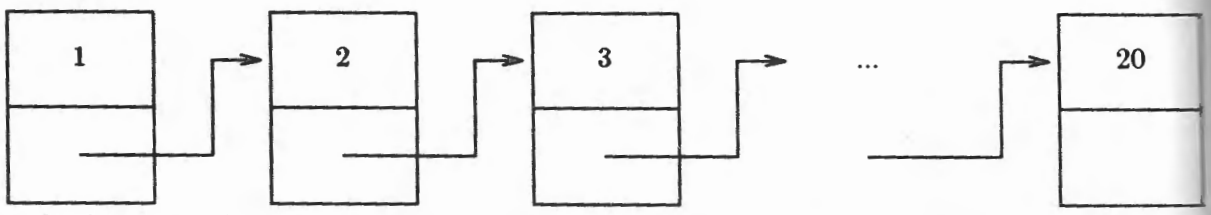
One way to create enough places to put the pointer values created by NEW statements is to situate the pointer within the target object that is created. For example,

```
TYPE
  Node = RECORD
           Value: INTEGER;
           Next: ↑Node
         END;
```

is a record containing a pointer to its own type. The identifier `Node` is used within its own definition, one of the rare occurrences in Pascal where definition need not precede use. In fact, a type identifier could be associated with `↑Node` before `Node` was defined:

```
TYPE
  NodePtr: ↑Node;
  Node = RECORD
           Value: INTEGER;
           Next: NodePtr
         END;
```

Using this data structure, the linked list of `INTEGER` values



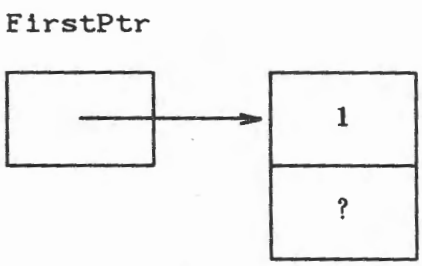
can be created. Access to the entire linked list will be provided by a pointer variable declared:

```
VAR
  FirstPtr: NodePtr;
```

The first list item and the access link can be implemented by:

```
BEGIN
  NEW(FirstPtr);
  FirstPtr↑.Value := 1
END
```

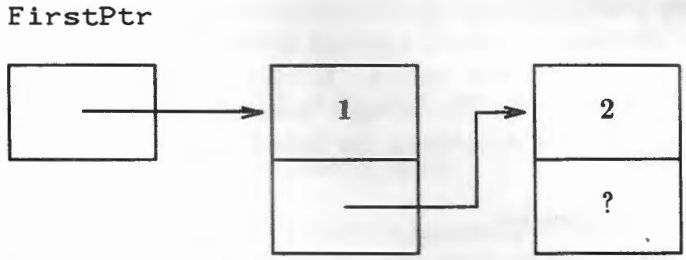
That is, `FirstPtr` points to a record of type `Node` whose `Value` field has value 1, and whose `Next` field is unknown:



To establish the second link, another NEW statement must be executed, but it should place the new pointer in the just-created record:

```
BEGIN
  NEW(FirstPtr↑.Next);
  FirstPtr↑.Next↑.Value := 2
END
```

The list is now:



The third list element could be created by the statement:

```
BEGIN
  NEW(FirstPtr↑.Next↑.Next);
  FirstPtr↑.Next↑.Next↑.Value := 3
END
```

Naming the place for each new pointer value beginning with `FirstPtr` will get longer and longer with each additional element, but a pointer variable `NewPtr` can be moved along the structure to make the reference easier. `NewPtr` is to point at the record just created, and to advance as the structure grows:

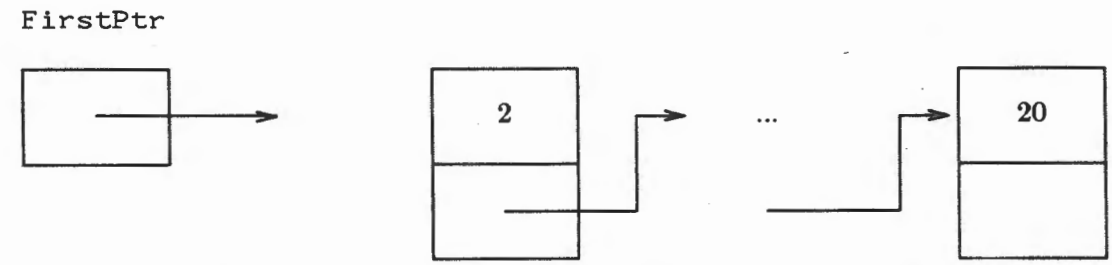
```
BEGIN
  {Create single element list with value 1}
  NEW(FirstPtr);
  FirstPtr↑.Value := 1;
  NewPtr := FirstPtr;
  FOR Index := 2 TO 20
  DO {Create element with value and position Index}
    BEGIN
      NEW(NewPtr↑.Next);
      NewPtr := NewPtr↑.Next;
      NewPtr↑.Value := Index
    END;
  NewPtr↑.Next := NIL
END
```

The final statement ends the list by assigning `NIL` to the last pointer field.

When pointers are heavily used it is important to reclaim storage that was once pointed at, but is no longer. Careful programming is necessary to avoid creating garbage while reclaiming the storage occupied by list structures. In the preceding example,

```
DISPOSE(FirstPtr)
```

returns its target's storage for reuse. That target is the first element in the list, containing the pointer to the remaining elements. However, these elements are not released, leaving the situation:



Thus 19 elements of the list remain as garbage, and `FirstPtr` is a dangling reference. In order to reclaim the storage occupied by the list structure, more care is required. The list must be traversed and a DISPOSE statement executed for each pointer, in the proper order so that nothing escapes.

20.2.3 Insertion Sort

The insertion sort program developed in Section 20.1.2 can be reimplemented using pointers in place of an array. The new implementation lifts the restriction that an array large enough to contain all expected records be available before reading the first record. In this implementation, records are added as needed and no extra records are created. The integer `Next` field of the array records is replaced by a `Next` field which is a pointer to the records of the linked list.

Design Part 2

```
PROGRAM InsertionSort(INPUT,OUTPUT);
{sort the characters in INPUT into ascending order}
  TYPE
    NodePtr = ↑Node;
    Node = RECORD
             Next: NodePtr;
             Key: CHAR
           END;
  VAR
    FirstPtr, NewPtr: NodePtr;
  BEGIN {InsertionSort}
    FirstPtr := NIL;
    WHILE NOT EOLN
    DO
      BEGIN
        NEW(NewPtr);
        READ(NewPtr↑.Key);
        {link NewPtr↑ into existing list}
      END;
    {print list starting with FirstPtr↑.Key}
  END.  {InsertionSort}
```

Two new `NodePtr` variables, `Prev` and `Curr` are added to obtain pointers to the elements which will be on either side of the new element after the linking is completed.

Design Part 2.1

```
{link NewPtr↑ into existing list}
Prev := NIL;
Curr := FirstPtr;
{find values for Prev and Curr (if they exist) such that
 Prev↑.Key <= NewPtr↑.Key <= Curr↑.Key}
NewPtr↑.Next := Curr;
IF Prev = NIL
THEN
  FirstPtr := NewPtr
ELSE
  Prev↑.Next := NewPtr
```

To move down the list searching for the correct position, the pointer `Curr` that began at `FirstPtr` is moved by repeatedly assigning it the `Next` field from the record to which it points:

Design Part 2.1.1

```
{find values for Prev and Curr (if they exist) such that
 Prev↑.Key <= NewPtr↑.Key <= Curr↑.Key}
Found := FALSE;
WHILE (Curr <> NIL) AND NOT Found
DO
  IF NewPtr↑.Key > Curr↑.Key
  THEN
    BEGIN {move to next element}
      Prev := Curr;
      Curr := Curr↑.Next
    END
  ELSE
    Found := TRUE;
```

Finally, the elements are printed by making NewPtr reference each list element in sorted order.

Design Part 2.2

```
{print list starting with FirstPtr↑.Key}
NewPtr := FirstPtr;
WHILE NewPtr <> NIL
DO
  BEGIN
    WRITE(NewPtr↑.Key);
    NewPtr := NewPtr↑.Next
  END;
WRITELN
```

20.2.4 A Binary Tree Sort

2-trees are usually called *binary trees,* and one can be used to implement a clever sorting algorithm in which the sorting is all done by tree traversal. The idea is to store the data and two pointers at each node, and to place data below a node in the left subtree if it falls earlier in sequence, or in the right subtree if it falls later in sequence. Once data are inserted according to this scheme, they can be retrieved in sorted order by a depth-first leftmost traversal of the tree. The data structure is:

```
TYPE
  Tree   = ↑NodeType;
  NodeType = RECORD
               Ch: CHAR;
               LLink, RLink: Tree
             END;
```

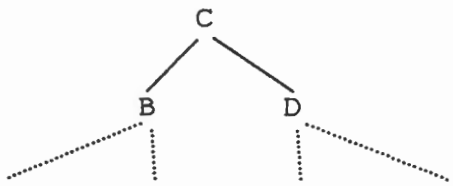
A character *C* is placed in the tree as follows:

> Starting at the root, *C* is compared with the character stored at a node. If it is the same or less, the tree is followed to the left, otherwise to the right. When the link to be followed is NIL, a new node is created with *C* as its data value.

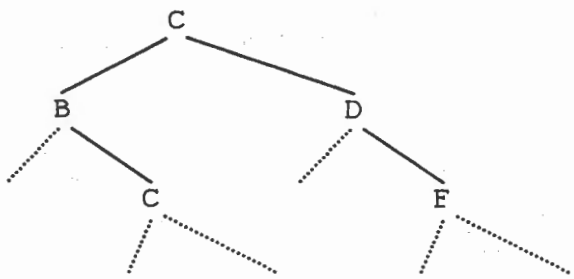
An elegant recursive procedure does the insertion:

```
PROCEDURE Insert(VAR Ptr: Tree; Data: CHAR);
{Insert the value of Data into the tree rooted at Ptr.
 Maintain the lexicographic ordering of the tree.}
  BEGIN {Insert}
    IF Ptr = NIL
    THEN
      BEGIN {Create leaf with value Data}
        NEW(Ptr);
        Ptr↑.Ch := Data;
        Ptr↑.LLink := NIL;
        Ptr↑.RLink := NIL
      END
    ELSE
      IF Ptr↑.Ch > Data
      THEN
        Insert(Ptr↑.LLink,Data)
      ELSE
        Insert(Ptr↑.RLink,Data)
  END; {Insert}
```

For example, the tree corresponding to the input string †**CBD**† is:



(The dotted links represent NIL pointers.) **C** was first placed at the root. Then **B** went left and **D** right. Suppose now that another **C** arrives, followed by **F**. The result will be:



Printing the characters in alphabetic order is accomplished by traversing the tree in the proper order. The rule for any node is to print everything in its left subtree, then the contents of the node, then everything in its right subtree. This also has a concise recursive implementation:

```
PROCEDURE PrintTree(Ptr: Tree);
{Print the values of the tree rooted at pointer.}
  BEGIN {PrintTree}
    IF Ptr <> NIL
    THEN
      BEGIN
        PrintTree(Ptr↑.LLink);
        WRITE(Ptr↑.Ch);
        PrintTree(Ptr↑.RLink)
      END
  END {PrintTree};
```

These two procedures are the backbone of a program that sorts a file of characters (ignoring line boundaries and special characters):

```
PROGRAM TreeSort(INPUT,OUTPUT);
{sort the characters in INPUT into ascending order}
  TYPE
    Tree  = ↑NodeType;
    NodeType = RECORD
                 Ch: CHAR;
                 LLink, RLink: Tree
               END;
  VAR
    Root: Tree;
    Ch: CHAR;
  {Include Insert and PrintTree}
  BEGIN {TreeSort}
    Root := NIL;
    WHILE NOT EOF
    DO
      BEGIN
        READ(Ch);
        WRITE(Ch);
        IF Ch IN ['A'..'Z','a'..'z','0'..'9']
        THEN
          Insert(Root,Ch);
      END;
    WRITELN;
    PrintTree(Root)
  END {TreeSort}.
```

Execution

```
INPUT :bcd
       bcefga
OUTPUT:bcd bcefga
       abbccdefg
```

20.2.5 Pointers, Functions, and Abstract Types

Pascal functions can return pointer values as well as scalar values.

<result type> ::= <simple type identifier> | <pointer type identifier>

This facility can be used to return aggregate values by allocating the value within the function and returning a pointer to the value as the result of the function.

In developing HarmonicSeries in Section 18.2.3,

```
{Sum := Sum + 1/Ctr)}
```

was refined to:

```
MakeRat(R,1,Ctr);    {R := 1/Ctr}
AddRat(Sum,Sum,R);   {Sum := Sum + R}
```

rather than:

```
Sum := AddRat(Sum,MakeRat(1,Ctr))
```

because a Rational value was represented as a RECORD, and functions cannot return RECORD results. The more natural functional statement could be used if the Rational type were a *pointer* to the record, which functions could return. The appropriate declarations are:

```
TYPE
  RatRec =
    RECORD
      Num, Den: INTEGER
    END;
  Rational = ↑RatRec;
```

Now the solution to HarmonicSeries (shown without the implementation of the Rational module) can be more naturally expressed:

```
PROGRAM HarmonicSeries (INPUT, OUTPUT);
{There is an integer n in INPUT,
  (n>0 --> Sum := 1 + 1/2 + 1/3 + 1/4 + ... + 1/n) |
  (n<=0 --> Sum := 0) }
  TYPE
    RatRec =
      RECORD
        Num, Den: INTEGER
      END;
    Rational = ↑RatRec;
  VAR
    Terms: INTEGER;
    Ctr: 0..MAXINT;
    Sum: Rational;
  BEGIN {HarmonicSeries}
    READ(Terms);
    Ctr := 0;
    {Sum := 0/1;}
    Sum := MakeRat(0,1);
    {WRITELN('Term=  0, Sum=',Sum)}
    WRITE('Term=  0, Sum=');
    WriteRat(OUTPUT,Sum);
    WRITELN;
    WHILE Ctr < Terms
    DO
      BEGIN
        Ctr := Ctr + 1;
        {Sum := Sum + 1/Ctr;}
        Sum := AddRat(Sum,MakeRat(1,Ctr));
        {WRITELN('Term= ', Ctr:2, ', Sum=', Sum)}
        WRITE('Term= ', Ctr:2, ', Sum=');
        WriteRat(OUTPUT,Sum);
        WRITELN
      END
```

```
END.  {HarmonicSeries}
```

The module operations must be implemented so that they return appropriate pointers of type `Rational`. A local variable of type `Rational` is declared in an operation that must return a pointer, a `RatRec` record and the `Rational` pointer to it are allocated using this local variable, and its result is assigned as the function's value. A local variable must be used instead of the function's name because NEW statements are not permitted with a function identifier. For the function `AddRat` the new local variable is `R` and the changes are:

```
FUNCTION AddRat(Op1, Op2: Rational): Rational;
  {abs: <AddRat> := <Op1 + Op2> }
  {con: <AddRat↑.Den,AddRat↑.Num> :=
          <Op1↑.Den * Op2↑.Den,
           Op1↑.Num * Op2↑.Den + Op1↑.Den * Op2↑.Num> }
  VAR
    R: Rational;
  BEGIN {AddRat}
    NEW(R);
    R↑.Num := Op1↑.Num * Op2↑.Den
              + Op1↑.Den * Op2↑.Num;
    R↑.Den := Op1↑.Den * Op2↑.Den;
    AddRat := R
  END;  {AddRat}
```

Using pointers to represent rational numbers really hides information about the implementation since users not only cannot tell how rationals are represented, but also cannot tell whether they are aggregates or scalars. Of course, this advantage is not without cost because accessing a rational number requires following a pointer in addition to the usual variable reference.

20.2.6 Exercises

20.2.1 Each Pascal fragment below contains the identifier `Qu`. You are to deduce a possible type of `Qu` from the context in each separate case. Express the answer as a type specification for the declaration:

```
VAR
  Qu: ?
```

Replace the question mark with the type required of `Qu`. For example, if you decide that one answer is a subrange of `CHAR` between **3** and **6**, fill in:

```
VAR
  Qu: '3'..'6'
```

If there are several possible types, explain briefly.

a) `Qu↑ := 'X'`

b) `IF Qu <> NIL THEN WRITELN('Continue')`

20.2.2 Give a type declaration for *T* to make the following Pascal fragment legal:

```
VAR
  P: T;
BEGIN
  P↑↑.Q := 'X'
END
```

20.2.3 Suppose that a linked list of `INTEGER` values is to be created, and the choice is to be made between an array with `INTEGER` subscripts and a pointer implementation. Explain why the pointers are "safer" in that it is impossible to confuse the values with the pointers, but that in the array implementation this confusion is allowed.

20.2.4 Suppose that for a particular Pascal machine, the DISPOSE statement sets the pointer value involved to `NIL`. Explain how it is still possible to create dangling references.

20.2.5 The collection {↑1, ↑2, ...} was used for pointer values in an execution state where several pointers *of the same type* were used. Does this one collection serve for all pointers of all possible types at once, or must a different one be used for each type? Explain.

20.2.6 For the variable `First`:

```
TYPE
  RefInt = ↑INTEGER;
VAR
  First: ↑RefInt
```

that might be the beginning of a list of *pointers* (that might themselves each point to a list of integers), write Pascal statements that create the value 73 and the pointers to connect `First` to it. Show a diagram and the contents of an execution state at each point in the process.

20.2.7 Write a fragment of Pascal code that properly releases the 20-element list at the end of Section 20.2.2.

20.2.8 Write a fragment of Pascal code that could be used with `InsertionSort` (Section 20.2.3) to print the list in *reverse* order (in place of Design Part 2.2).

a) Use an iterative method.

b) Use a recursive method.

20.2.9 The design of `TreeSort` (Section 20.2.4) is done "bottom-up" instead of in the usual way. The essential procedures are developed first, then combined to solve the problem.

a) Give a design using the usual top-down method.

b) For small problems there doesn't appear to be much difference between top-down and bottom-up design. Explain how bottom-up design could go wrong for a large problem.

20.2.10 Identify as many "typical" cases as you can in the `TreeSort` problem, without considering the implementation in Section 20.2.4. These are cases in which the action required might be expected to be the same for many different actual patterns in the data structure. For example, it is a typical case when the character to be inserted falls to the left of a node, and the node's left pointer is `NIL`.

a) Make a list of the cases identified, and of input test data that would excite each case. (Consider both inserting and printing operations.)

b) Now examine the actual implementation of `TreeSort`. Which of the cases in a) are really distinct in the sense that the program might work for one but fail for another?

20.2.11 Write a procedure to be used with `TreeSort` that determines the *height* of the tree. The height is defined as the longest path from the root to a leaf (a node with two `NIL` pointers).

20.2.12 Give the implementations for `MakeRat` and `WriteRat` using the pointer structure of Section 20.2.5.

20.3 Chapter Summary

Whether they are represented as arrays or pointers, linked structures trade increased access time (using an extra memory reference to obtain a value) for decreased copying costs (copying a reference to an object rather than the object itself). While the size of a linked structure represented by an array is static, pointers and the NEW operation provide programmers with an exact means of representing objects whose size varies during execution.

The level of indirection provided by pointers makes them particularly well suited for representing abstract data types. Any object represented by a pointer can be returned as the value of a function or passed as a value parameter without incurring the penalty for copying the object itself.

CHAPTER 21

GOTO STATEMENTS AND STRUCTURED PROGRAMS

Chapter Preview

The presence of GOTO statements affecting the flow of control in a program makes it unstructured, and in most cases more difficult to understand and analyze. GOTO statements may have a place for dealing with exceptions and for improving program efficiency, but in most cases they must be eliminated for the sake of understanding.

A GOTO statement can transfer control to almost any statement in a program. Programs with GOTO statements can be difficult to understand because the transfers destroy the hierarchical, part-function structure of the other Pascal statements. As a result, even though a Pascal program with (any number of) GOTO statements has a program function, that program function cannot be calculated one statement at a time. Instead, a program with GOTO statements must be analyzed in larger pieces, considering many statements simultaneously. Because of this intertwining of statements, the analysis is more complex, more error-prone. In short, a program with GOTO statements can be a real "can of worms" to unravel, and though the analysis is theoretically possible, it is impractical.

21.1 The GOTO Statement

Preview

Execution of a GOTO statement causes a break in the normal statement-to-statement control flow. The next statement executed after a statement GOTO *l* is the one at label *l*. Some error conditions are best handled by escaping from the normal sequence with a GOTO.

The GOTO statement provides a means of transferring control from the currently executing statement to another labeled statement in the program. Labels are unsigned integers containing from one to four digits, set off from the statement labeled by a colon.

<statement> ::= <unlabeled statement> | <label> : <unlabeled statement>

<unlabeled statement> ::= <simple statement> | <structured statement>

<simple statement> ::= <assignment statement> | <procedure statement>
| <null statement> | <goto statement> | <case statement>
| <READ statement> | <WRITE statement>

<goto statement> ::= GOTO <label>

<label> ::= <digit sequence>

<structured statement> ::= <BEGIN statement> | <IF statement>
| <repetitive statement> | <with statement>

<repetitive statement> ::= <WHILE statement> | <repeat statement>
| <for statement>

Pascal requires that labels be explicitly declared in a label declaration at the start of the block that contains the statement bearing the label.

<program> ::= <program heading> ; <block> .

<program heading> ::= PROGRAM <identifier> (<identifier list>)

<block> ::= <label declaration part> <constant definitions>

<type definitions> <variable declarations>

<procedure and function declarations> <BEGIN statement>

<label declaration part> ::= LABEL <label list> ; ¦

<label list> ::= <label list> , <label> ¦ <label>

As with other declarations, duplicate declarations of labels within a unit of scope is forbidden although labels may be reused in disjoint or inner units of scope.

21.1.1 IF and WHILE Statements with GOTO

The GOTO statement is something of an anachronism in Pascal, a statement intended more to take advantage of a computer's execution capabilities than to express program designs. In assembly languages and in some versions of FORTRAN and BASIC, the Pascal IF and WHILE statements are not available, and GOTO statements must be used to simulate them as follows:

```
IF B                    IF B
  THEN                    THEN
    S                       GOTO 1;
  ELSE                  T; GOTO 2;
    T                   1: S;
                        2:
```

```
WHILE B                 1: IF NOT B
DO                         THEN
  S                          GOTO 2
                           S;
                           GOTO 1;
                        2:
```

Few programmers would choose to give up the Pascal IF and WHILE statements in favor of these alternatives. The GOTO statements destroy the textual structure of the program intended by the programmer. Programs with GOTO statements are difficult to read because control can be transferred either forward or backward, and to learn the conditions under which a labeled statement may be executed, it is necessary to consider each GOTO statement that references its label. However, the GOTO statement is a natural way to express an *intended* break in the control pattern, most often to escape from an iteration early.

21.1.2 Escaping from an Error Condition

Because the WHILE statement continues to repeat its BEGIN statement so long as the condition is met, it is difficult to abort an iteration early. The situation arises in searching. Suppose that the array `Line` contains character values, and index positions 1 to 100 are to be searched for **F**, and its subscript returned. It is natural to write

```
FOR Sub := 1 TO 100
DO
  IF Line[Sub] = 'F'
  THEN
    {What??}
```

The programmer needs to terminate the iteration early, and so may resort to using a BOOLEAN loop control and a WHILE statement:

```
Found := FALSE;
Sub := 1;
WHILE (NOT Found) AND (Sub <= 100)
DO
  BEGIN
    IF Line[Sub] = 'F'
    THEN
      Found := TRUE;
    Sub := Sub + 1
  END;
IF Found
THEN
  WRITELN('F found at position ', Sub:3)
ELSE
  WRITELN('F missing from list.')
```

With a GOTO another version using a FOR statement can duplicate this behavior:

```
FOR Sub := 1 TO 100
DO
  IF Line[Sub] = 'F'
  THEN
    GOTO 99;
IF FALSE
THEN
  99: WRITELN('F found at position ', Sub:3)
ELSE
  WRITELN('F missing from list.')
```

As another example, consider the problem of reading two matrices (i.e., two-dimensional arrays) of integers from input, perhaps to compute their product as in Section 19.4.2. Suppose that the matrices are given a row at a time, in sequence. For example, the matrices:

$$\begin{bmatrix} 3 & 7 & 2 \\ 1 & 4 & 9 \end{bmatrix} \text{ and } \begin{bmatrix} 2 & 4 \\ 3 & 1 \\ 8 & 2 \end{bmatrix}$$

would appear as the input string:

†3 7 2 1 4 9 2 4 3 1 8 2†.

In this example, 12 integers are expected. If only 7 are supplied, the first matrix is defined, but not the second. The following partial design deals with the possibility of insufficient data. A BOOLEAN variable DataOK is used to indicate that enough data were present to define both matrices.

Design Part 1

```
PROGRAM Matrices(INPUT, OUTPUT);
  CONST
    Sub1 = 2;
    Sub2 = 3;
    Sub3 = 2;
  TYPE
    Matrix1 = ARRAY[1..Sub1, 1..Sub2] OF INTEGER;
    Matrix2 = ARRAY[1..Sub2, 1..Sub3] OF INTEGER;
  VAR
    M1: Matrix1;
    M2: Matrix2;
    I:  1..Sub1;
    J:  1..Sub2;
    K:  1..Sub3;
    DataOK: BOOLEAN;
BEGIN {Matrices}
  IF NOT EOF
  THEN
    {read and echo M1};
  IF NOT EOF
  THEN
    {read and echo M2};
  {DataOK := M1 and M2 both full};
  IF DataOK
  THEN
    {compute using M1 and M2}
END {Matrices}.
```

Design Parts 1.1 and 1.2 are very similar—nested WHILE statements are used to fill the matrices one row at a time. Integers are being read from a text file (which includes an line marker after the last integer), so care is required not to read past the end of file. As described in Section 14.3.2, it is necessary to skip over blanks that might end the file, and the procedure `SkipBlanks` does this.

Design Part 1.1

```
BEGIN {read and echo M1}
  I := 1;
  WHILE (NOT EOF) AND (I <= Sub1)
    DO
    BEGIN
      J := 1;
      WHILE (NOT EOF) AND (J <= Sub2)
      DO
        BEGIN
          SkipBlanks;
          IF EOF
          THEN
            WRITELN('** Too little data for M1 **')
          ELSE
            BEGIN
              READ(M1[I,J]);
              WRITE(M1[I,J]);
              J := J + 1
            END
        END;
```

```
        I := I + 1
      END;
    WRITELN
END;
```

Design Part 1.2

```
BEGIN {read and echo M2}
  J := 1;
  WHILE (NOT EOF) AND (J <= Sub2)
  DO
    BEGIN
      K := 1;
      WHILE (NOT EOF) AND (K <= Sub3)
      DO
        BEGIN
          SkipBlanks;
          IF EOF
          THEN
            WRITELN('** Too little data for M2 **')
          ELSE
            BEGIN
              READ(M2[J,K]);
              WRITE(M2[J,K]);
              K := K + 1
            END
        END;
      J := J + 1
    END;
  WRITELN
END;
```

M1 and M2 are both full if the last WHILE statement terminated because the value of J passed that of Sub2.

Design Part 1.3

```
{DataOK := M1 and M2 both full}
DataOK := J = Sub2+1;
```

While this treatment of insufficient data is satisfactory, it can be optimized by use of a single label and two GOTO statements to handle just the case of insufficient data. The label declaration should be inserted just after the program header, and the entire program must be shown at once to see both the label and the GOTO statements that target it:

Design Part 2

```
PROGRAM Matrices2(INPUT, OUTPUT);
  LABEL
    1;
  {Insert the declarations from program Matrices}
BEGIN {Matrices2}
  FOR I := 1 TO Sub1
  DO
    FOR J := 1 TO Sub2
    DO
      BEGIN
        SkipBlanks;
        IF EOF
        THEN
          BEGIN
            WRITELN('** Too little data for M1 **');
            GOTO 1
          END;
        READ(M1[I,J])
      END;
  FOR J := 1 TO Sub2
  DO
    FOR K := 1 TO Sub3
    DO
      BEGIN
        SkipBlanks;
        IF EOF
        THEN
          BEGIN
            WRITELN('** Too little data for M2 **');
            GOTO 1
          END;
        READ(M2[J,K])
      END;
  Mult(M1,M2,M3); {M3 := M1 * M2}
  FOR I := 1 TO Sub1
  DO
    BEGIN
      FOR K := 1 TO Sub3
      DO
        WRITE(M3[I,K]);
      WRITELN
    END;
  1:
END {Matrices2}.
```

This second program is shorter than the first, and perhaps easier to understand. Some programmers use GOTO statements only to handle error processing so that readers of a program can ignore these statements when first trying to understand the "normal" cases treated by the program. However, the example shows a drawback of using GOTO statements even in this limited way: unless the label and the statements that reach it can all be viewed at once, the clarity is lost. In many cases this would mean that the program cannot be split into sufficiently small pieces to retain intellectual control of its design.

21.1.3 Transfers into Composite Statements

Although it is possible to jump out of a statement (e.g., the FOR statements in `Matrices2` above), it is illegal to jump into some composite statements from outside. (The error is often not detected on particular Pascal machines.) An example of such an illegal jump is shown below:

```
GOTO 1; {Illegal because inside FOR statement}
FOR I := 1 TO N
DO
  BEGIN
    S1;
    1:S2
  END
```

The restriction forbidding the jump to label `1` is sensible, since by avoiding the header of the FOR statement, it transfers to a place where the index variable `I` might be used, but has not been initialized. Thus there is no similar restriction on transfer into a WHILE statement, where the an index value, if any, is under explicit control. Similarly, one may not jump into the BEGIN statement of a procedure or function from outside, but it is all right to jump into the main BEGIN statement of a program from anywhere. The reason is that the program header has necessarily been encountered to set up the main BEGIN statement, but a jump into a procedure avoids its header, and local variables and parameters might not be properly set up.

GOTO statements not only change the flow of control, but may also change the execution state. In the following example, the jump from `Q` into the BEGIN statement of `P` removes the local A from the state:

```
PROGRAM P (OUTPUT);
  LABEL
    18;
  VAR
    A: INTEGER;
  PROCEDURE Q;
    VAR
      A: INTEGER;
    BEGIN {Q}
      A := 1;
      GOTO 18; {exit Q and deallocate A}
      WRITELN(A)
    END; {Q}
  BEGIN {P}
    A := 0;
    Q;
  18:WRITELN(A)
  END {P}.
```

Execution

```
OUTPUT:          0
```

This example and the others in this section make it clear that the analysis techniques of the program calculus would have to be modified to deal with programs containing GOTO statements.

21.1.4 Exercises

21.1.1 A label must be a digit sequence whose length is 4 or less, yet the syntax given in Section 21.1 does not control the length. Is this necessarily done with a context condition, or can syntax equations be written for exactly those sequences that *are* labels?

21.1.2 What mistakes involving labels and GOTO statements occur in the following program?

```
PROGRAM TooManyGotos (INPUT,OUTPUT);
  LABEL 10, 20;
  PROCEDURE Inner;
    LABEL 30;
    BEGIN {Inner}
      20: GOTO 30;
      30: GOTO 10
    END;  {Inner}
  BEGIN {TooManyGotos}
    GOTO 20;
    GOTO 30;
    10: WRITELN('That''s enough')
  END.  {TooManyGotos}
```

21.1.3 Rewrite the following Pascal fragment so that no GOTO statements appear but the program performs the same computation.

```
LABEL 10, 20;
BEGIN
  10: Statement1;
  IF B
  THEN
    GOTO 20;
  Statement2;
  GOTO 10;
  20:
END
```

Does another program with the same computation and no GOTO statements exist that does not repeat either `Statement1` or `Statement2`?

21.1.4 One way to attempt to bring GOTO statements under the control of the program calculus is to treat them like procedure statements. That is, encountering a label is like encountering a procedure declaration, and transferring to that label is like calling the procedure. Investigate this idea for a program consisting of a WHILE statement rewritten using two GOTO statements as in Section 21.1.1.

21.2 Analysis of Programs with GOTO Statements

Preview

The analysis techniques of the program calculus depend on having structured programs containing no GOTO statements. When a program is unstructured it can be converted to a structured program for analysis.

A Pascal program with the hierarchical part-function structure of the program calculus is called a *structured program*. Conversely, a program that lacks this structure (because it uses unrestrained GOTO statements) is called an *unstructured program*. The program calculus presented in this book applies only to structured programs, and all methods known for calculating the meaning of unstructured programs are much more difficult to use. Thus the way to analyze an unstructured program is to first convert it to structured form.

21.2.1 Structuring an Unstructured Program

A *closed statement* is a Pascal statement whose text contains:

all labels for its GOTO statements, and

all GOTO statements for its labels.

That is, a closed statement may contain GOTO statements, but no GOTO statements may enter or leave it. An *open statement* is one that is not closed. The following statements are closed:

The BEGIN statement of a program that has no procedures or functions.

Any statement lacking both GOTO statements and labels.

A procedure or function can have GOTO statements without their corresponding labels, so its BEGIN statement is not necessarily closed. The analysis and verification of a program containing open procedures can be very difficult. However, by copying the procedure text in place of the call (suitably altered to deal with name conflicts and parameters), eventually the internal GOTO statements will be in the same BEGIN statement as their target labels. Therefore, without any loss in generality, we assume that the BEGIN statement of a program and of every procedure and function is closed.

Since unstructured programs can be much harder to understand than structured programs, it is natural to wonder also how much more they can do. Are there problems that can be solved using an unstructured program that have no structured solution? The answer may be surprising: unstructured programs can do no more than structured programs. The ease with which this can be shown may also be surprising.

Given an arbitrary unstructured program U, a method will be given for converting it into a structured program that duplicates the behavior of the original. The program is assumed to contain only GOTO statements and statements of CF Pascal; the additional cases for full Pascal are similar and are treated in the Exercises.

First, U may be assumed to be a program in which the main BEGIN statement and the BEGIN statements of all procedures and functions are closed, as explained above. Since each BEGIN statement will be treated in the same way, the case of just a main program will be treated without loss of generality.

The BEGIN statement of U will be transformed to structured form. Make up a label for each statement of U that lacks one, so that each statement is labeled, say with the labels L_1, L_2, ..., L_k, for the k statements in the BEGIN statement of U. Label a null statement at the final `END` L_{k+1}. Declare an `INTEGER` variable not in U, which will here be taken to be `LabelValue`. Now, replace the BEGIN statement of U with a new BEGIN statement of the form:

```
BEGIN
  LabelValue := Init;
  WHILE LabelValue <> Exit
  DO
    CASE LabelValue OF
      L1: ...
      L2: ...
      ...
      Lk: ...
    END
END
```

where `Init` is a constant with the value of the first label L_1 in U and `Exit` is similarly the value of the final label L_{k+1}.

The statements to be inserted in the CASE statement are modifications of the ones from U, depending on the form of the original statement.

When the statement S with label L in U is a null statement, an assignment statement, a READ or WRITE statement, or a procedure statement, place the following at L within the CASE statement:

```
BEGIN
  S;
  LabelValue := M
END
```

where M is the label of the statement following S in the original program U.

When the statement in U is an IF statement:

```
L: IF B
     THEN
       T: ...
     ELSE
       E: ...
```

Then within the CASE statement at label L place:

```
IF B
THEN
  LabelValue := T
ELSE
  LabelValue := E
```

and similarly if there is no ELSE part to the IF statement. (The statements labeled T and E will be transformed and also appear in the CASE statement.)

When the statement in U is a WHILE statement:

```
L: WHILE B
     DO
       BEGIN
         D: ...
       END;
E:
```

Then within the CASE statement at label L place:

```
IF B
THEN
  LabelValue := D
ELSE
  LabelValue := E
```

The easiest case is the one for which the whole scheme was invented. When the statement in U is:

```
L: GOTO M
```

the statement at L within the CASE is:

```
LabelValue := M
```

The program thus constructed from U is structured, since no GOTO statements were introduced, and those of U were all replaced. The control flow of U is exactly reproduced in the sense that each test performed in U and each statement executed in U are done in the same order, but between statements `LabelValue` is computed, and the WHILE and CASE statements direct the execution based on this value. However, since `LabelValue` was not present in U, these additional calculations cannot alter any results U produced, and hence the structured program will exactly reproduce any input-output behavior of U.

The structured program produced from an unstructured original is not an example of good programming style. However, it can be analyzed using the program calculus, and it may prove possible to transform it into a simpler equivalent form.

21.2.2 A Structuring Example

In order to illustrate the structuring construction, consider the program:

```
PROGRAM GreatCalculationDone(INPUT, OUTPUT);
  LABEL
    10, 20, 30, 40, 50, 60, 70, 80, 90, 100;
  VAR
    I1, I2: INTEGER;
  BEGIN
    10: READ(I1, I2); GOTO 20;
    20: IF (I1 > 0) AND (I2 > 0) THEN GOTO 80
        ELSE GOTO 100;
    30: IF I1 < I2 THEN GOTO 70 ELSE GOTO 50;
    40: IF I1 = I2 THEN GOTO 90 ELSE GOTO 30;
    50: IF I1 > I2 THEN GOTO 60 ELSE GOTO 40;
    60: I1 := I1 - I2; GOTO 40;
    70: I2 := I2 - I1; GOTO 40;
    80: WRITE('DATA IS', I1:3, I2:3); GOTO 40;
    90: WRITELN(', ANSWER IS', I1:3); GOTO 10;
    100:
  END.
```

The program uses GOTO statements to present a neat appearance, with all READ, IF, assignment, and WRITE statements grouped together. But what the program does—its program function—is another matter. If its WRITE statements are to be believed, it takes two input integers and produces a third integer as result. But what is the result? If `GreatCalculationDone` were structured, its program function could be obtained step by step using techniques of the program calculus. But that will not work here. Instead, the entire set of statements is locked together with all those GOTO statements, so that no part can be separated out for analysis and study. `GreatCalculationDone` is a real can of worms. It doesn't help very much to perform trial executions, or to work out their execution tables. One quickly learns that the program echos its input in pairs, and terminates when either input is not positive, but it isn't obvious how the result is obtained.

The structured version of `GreatCalculationDone` is obtained by adding labels to the unlabeled statements and carrying out the construction of the last section. Since there are no procedures, only the BEGIN statement of the program must be structured. First the additional labels:

```
PROGRAM GreatCalculationDone1(INPUT, OUTPUT);
  LABEL
    10, 15, 20, 23, 27, 30, 33, 37, 40, 43, 47,
    50, 53, 57, 60, 65, 70, 75, 80, 85, 90, 95,
    100;
  VAR
    I1, I2: INTEGER;
  BEGIN
    10: READ(I1, I2); 15: GOTO 20;
    20: IF (I1 > 0) AND (I2 > 0) THEN 23: GOTO 80
        ELSE 27: GOTO 100;
    30: IF I1 < I2 THEN 33: GOTO 70 ELSE 37: GOTO 50;
    40: IF I1 = I2 THEN 43: GOTO 90 ELSE 47: GOTO 30;
    50: IF I1 > I2 THEN 53: GOTO 60 ELSE 57: GOTO 40;
    60: I1 := I1 - I2; 65: GOTO 40;
    70: I2 := I2 - I1; 75: GOTO 40;
    80: WRITE('DATA IS', I1:3, I2:3); 85: GOTO 40;
    90: WRITELN(', ANSWER IS', I1:3); 95: GOTO 10;
```

```
    100:
  END.
```

Introducing the new variable L1 for the label values, the structured version is:

```
PROGRAM GreatCalculationDone2(INPUT, OUTPUT);
  VAR
    I1, I2, L1: INTEGER;
  BEGIN
    L1 := 10;
    WHILE L1 <> 100
    DO
      CASE L1 OF
        10: BEGIN READ(I1, I2); L1 := 15 END;
        15: L1 := 20;
        20: IF (I1 > 0) AND (I2 > 0) THEN L1 := 23
            ELSE L1 := 27;
        23: L1 := 80;
        27: L1 := 100;
        30: IF I1 < I2 THEN L1 := 33 ELSE L1 := 37;
        33: L1 := 70;
        37: L1 := 50;
        40: IF I1 = I2 THEN L1 := 43 ELSE L1 := 47;
        43: L1 := 90;
        47: L1 := 30;
        50: IF I1 > I2 THEN L1 := 53 ELSE L1 :=57;
        53: L1 := 60;
        57: L1 := 40;
        60: BEGIN I1 := I1 - I2; L1 := 65 END;
        65: L1 := 40;
        70: BEGIN I2 := I2 - I1; L1 := 75 END;
        75: L1 := 40;
        80: BEGIN
              WRITE('DATA IS', I1:3, I2:3);
              L1 := 85
            END;
        85: L1 := 40;
        90: BEGIN
              WRITELN(', ANSWER IS', I1:3);
              L1 := 95
            END;
        95: L1 := 10
      END
  END.
```

Now that the program is structured, it can be simplified through a series of transformations. The first transformation eliminates cases that are just assignments to L1. A case like:

```
15: L1 := 20;
```

can be eliminated by changing all the assignment statements of the the form:

```
L1 := 15
```

to

```
L1 := 20
```

Thus the statement labeled 10 changes to:

```
10: BEGIN READ(I1, I2); L1 := 20 END;
```

and the statement labeled 15 is deleted. The result of this transformation is:

```
PROGRAM GreatCalculationDone3(INPUT, OUTPUT);
  VAR
    I1, I2, L1: INTEGER;
  BEGIN
    L1 := 10;
    WHILE L1 <> 100
    DO
      CASE L1 of
        10: BEGIN READ(I1, I2); L1 := 20 END;
        20: IF (I1 > 0) AND (I2 >0) THEN L1 := 80
            ELSE L1 := 100;
        30: IF I1 < I2 THEN L1 := 70 ELSE L1 := 50;
        40: IF I1 = I2 THEN L1 := 90 ELSE L1 := 30;
        50: IF I1 > I2 THEN L1 := 60 ELSE L1 := 40;
        60: BEGIN I1 := I1 - I2; L1 := 40 END;
        70: BEGIN I2 := I2 - I1; L1 := 40 END;
        80: BEGIN
              WRITE('DATA IS', I1:3, I2:3);
              L1 := 40
            END;
        90: BEGIN
              WRITELN(', ANSWER IS', I1:3);
              L1 := 10
            END
      END
  END.
```

Next, suppose a statement contains no assignment of its own label to L1. Then that statement can be inserted in place of every assignment of L1 to its label. For example,

```
20: IF (I1 > 0) AND (I2 > 0) THEN L1 := 80 ELSE L1 := 100
```

does not assignment 20 to L1. Therefore this IF statement can be substituted for

```
L1 := 20
```

everywhere, in particular in:

```
10: BEGIN READ(I1, I2); L1 := 20 END
```

to yield:

```
10: BEGIN
      READ(I1, I2);
      IF (I1 > 0) AND (I2 > 0)
      THEN
        L1 := 80
      ELSE
        L1 := 100
    END
```

Label 40 is one that could be eliminated in this way, but it would produce a four-fold duplication, so only the labels 30, 50, 60, 70, 80, and 90 are eliminated:

```
PROGRAM GreatCalculationDone4(INPUT, OUTPUT);
  VAR
    I1, I2, L1: INTEGER;
  BEGIN
    L1 :=10;
    WHILE L1 <> 100
    DO
      CASE L1 OF
        10: BEGIN
              READ(I1, I2);
              IF (I1 > 0) AND (I2 > 0)
              THEN
                BEGIN
                  WRITE('DATA IS', I1:3, I2:3);
                  L1 := 40
                END
              ELSE
                L1 := 100
            END;
        40: IF I1 = I2
            THEN
              BEGIN
                WRITELN(', ANSWER IS', I1:3);
                L1 := 10
              END
            ELSE
              IF I1 < I2
              THEN
                BEGIN
                  I2 := I2 - I1;
                  L1 := 40
                END
              ELSE
                IF I1 > I2
                THEN
                  BEGIN
                    I1 := I1 - I2;
                    L1 := 40
                  END
                ELSE
                  L1 := 40
      END
  END.
```

The statement at 40 can be simplified by observing that the most deeply nested IF condition is unnecessary because previous IF statements have ruled out the cases `I1=I2` and `I1<I2`. All possibilities in the statement end with the assignment to 40, so this can be factored out. These changes give:

```
40:IF I1 = I2
   THEN
     BEGIN
       WRITELN(', ANSWER IS', I1:3);
       L1 := 10
     END
   ELSE
     BEGIN
       IF I1 < I2
       THEN
         I2 := I2 - I1
       ELSE {I1 > I2}
         I1 := I1 - I2;
       L1 := 40
     END
```

It can now be seen that the statement at 40 will be repeatedly executed by the main CASE statement so long as the values of I1 and I2 differ. It can therefore be converted into a WHILE statement that does not use the outer CASE mechanism, but simply iterates until I1 and I2 are the same:

```
40:BEGIN
     WHILE I1 <> I2
     DO
       IF I1 < I2
       THEN
         I2 := I2 - I1
       ELSE
         I1 := I1 - I2;
     WRITELN(', ANSWER IS', I1:3);
     L1 := 10
   END
```

Since there are no longer any assignments L1 := 40 within the statement labeled 40, it can be substituted into case 10 in place of the assignment there. The CASE statement itself can now be removed, since only one case remains. The resulting program is:

```
PROGRAM GreatCalculationDone5(INPUT, OUTPUT);
  VAR
    I1, I2, L1: INTEGER;
  BEGIN
    L1 := 10;
    WHILE L1 <> 100
    DO
      BEGIN
        READ(I1, I2);
        IF (I1 > 0) AND (I2 > 0)
        THEN
          BEGIN
            WRITE('DATA IS', I1:3, I2:3);
            WHILE I1 <> I2
            DO
              IF I1 < I2
              THEN
                I2 := I2 - I1
              ELSE
                I1 := I1 - I2;
```

```
            WRITELN(', ANSWER IS', I1:3)
          END
        ELSE
          L1 := 100
      END
  END.
```

The variable L1 now only controls exit from the WHILE statement, and can be eliminated by changing the WHILE condition to

```
WHILE (I1 > 0) AND (I2 > 0)
```

which reflects the only way that the iteration terminates. There is one complication. The termination test follows a READ statement, but moved into the WHILE it would precede that READ. A way to deal with this is to duplicate the READ statement before the WHILE statement and move it to the end of the DO statement:

```
PROGRAM GreatCalculationDone6(INPUT, OUTPUT);
  VAR
    I1, I2: INTEGER;
  BEGIN
    READ(I1, I2);
    WHILE (I1 > 0) AND (I2 > 0)
    DO
      BEGIN
        WRITE('DATA IS', I1:3, I2:3);
        WHILE I1 <> I2
        DO
          IF I1 < I2
          THEN
            I2 := I2 - I1
          ELSE
            I1 := I1 - I2;
        WRITELN(', ANSWER IS', I1:3);
        READ(I1, I2)
      END
  END.
```

It is interesting to compare the practically inscrutable unstructured program GreatCalculationDone with the structured version GreatCalculationDone6. The latter program can be seen to calculate with successive pairs of input integers. In order to discover what the calculation is, it only remains to study the inner WHILE statement *W*:

```
WHILE I1 <> I2
DO
  IF I1 < I2
  THEN
    I2 := I2 - I1
  ELSE
    I1 := I1 - I2
```

It is not obvious at first glance that *W* ever terminates. However, in the IF statement, the greater of I1 and I2 is decremented by the lesser of these on each iteration. Thus one of the two values always decreases, and neither can go to zero, since they start out unequal. This decrementing process bounded below cannot continue indefinitely, so the iteration must terminate.

The termination argument shows that $\boxed{W}$ has the form:

```
(I1=I2 → ) |
(I1<>I2 AND I1>0 AND I2>0 → I1,I2 := g(I1,I2),g(I1,I2) )
```

for some function g. Two special cases are instructive. Suppose that I2 is initially greater than I1, and when I2 is repeatedly reduced by I1, the values are eventually the same. Then I2 began as a multiple of I1, and

$$g(I1,I2) = I1,\ I2 = k \times I1.$$

By symmetry,

$$g(I1,I2) = I2,\ I1 = k \times I2.$$

It would therefore be a good guess that g is the *GCD* function discussed in Section 18.2.4.

In the general case that neither variable is a multiple of the other, reducing I2 by a multiple of I1 until I1>I2 is equivalent to finding a $k \geq 1$ and an h, $0 < h < I1$, such that:

$$I2 - k \times I1 + h = I1.$$

That is,

$$I2 = k \times I1 + (I1 - h)$$

or in more familiar terms:

$$I2 = k \times I1 + r$$

where $k = I2 \div I1$ and r is the remainder. As the reduction process is repeated, first on one variable and then on the other, both are eventually reduced to a value that divides them both, and is the least such value. That is, W computes the GCD of I1 and I2.

21.2.3 Exercises

21.2.1 Eliminate the GOTO statement from the following program, distorting its structure as little as possible.

```
PROGRAM Search(INPUT,OUTPUT);
{Read a list of integer values from input and print
 the distinct values and the number of times they occur}
  LABEL
    10;
  CONST
    MaxSize = 10;
  TYPE
    IType = 1..MaxSize;
  VAR
    Vals: ARRAY [IType] OF INTEGER;
    Searches: ARRAY [IType] OF INTEGER;
    CurrSize: 0..MaxSize;
    Key: INTEGER;
    I: IType;
```

```
BEGIN {Search}
  CurrSize := 0;
  READ(Key);
  WHILE Key <> -1
  DO
    BEGIN
      FOR I := 1 TO CurrSize
      DO
        IF Vals[I] = Key
        THEN
          GOTO 10;
      I := CurrSize + 1;
      CurrSize := I;
      Vals[I] := Key;
      Searches[I] := 0;
   10:Searches[I] := Searches[I] + 1;
      READ(Key)
    END;
  WRITELN('Values',' Searches');
  FOR I := 1 TO CurrSize
  DO
    WRITELN(Vals[I]:6,Searches[I]:9)
END.  {Search}
```

21.2.2 Remove the GOTO statement from the following program, distorting its structure as little as possible.

```
PROGRAM Format(INPUT,OUTPUT);
{Read INPUT and produce formatted text in OUTPUT.
 Translate // to a new line and / to a tab character.
 Add an extra space after each period.}
 LABEL
   10;
 CONST
   Escape = '/';
 BEGIN {Format}
   WHILE NOT EOF
   DO
     BEGIN
       IF INPUT↑ = Escape
       THEN
         BEGIN
           GET(INPUT);
           IF NOT EOF
           THEN
             IF INPUT↑ = Escape
             THEN
               BEGIN
                 WRITELN;
                 GOTO 10
               END
             ELSE
               WRITE(CHR(9))
         END;
       OUTPUT↑ := INPUT↑;
```

```
        PUT(OUTPUT);
        IF INPUT↑ = '.'
        THEN
          WRITE(' ');
      10:GET(INPUT)
      END;
    WRITELN
  END.  {Format}
```

Execution

```
INPUT :one. two. three.///four/five/six.
OUTPUT:one.  two.  three.
             four    five    six.
```

21.2.3 Show that if every procedure and function of a Pascal program has a closed BEGIN statement, then the program's main BEGIN statement is necessarily also closed.

21.2.4 In the construction of a structured program from an unstructured one in Section 21.2.1, explain why the *statements* from within IF and WHILE statements are not inserted at the labels for the IF and WHILE.

21.2.5 In the construction of Section 21.2.1, suppose that the labels placed on the original program statements are all in numerical order, increasing throughout the program text.

a) Explain why the original program will not necessarily produce the same results if the order of statements is changed by moving a complete labeled statement.

b) Explain why moving labeled statements within the CASE statement of the structured version does not change the program's meaning.

21.2.6 The construction of Section 21.2.1 covers only the CF Pascal language with GOTO statements added. This exercise is to show that the construction can be extended to all of Pascal.

a) Extend the construction to cover CASE statements.

b) Extend the construction to cover REPEAT statements.

c) Extend the construction to cover FOR statements.

d) Explain why the additional data types of D and O Pascal do not enter into the construction.

21.2.7 Prove that for every CF Pascal program, an equivalent program can be written using exactly *one* WHILE statement. What constructions not in CF Pascal are *required* to write this equivalent?

21.2.8 Suppose a program is structured, but the scheme for structuring is applied to it anyway. Compare the quality of the original with that of the new structured program.

21.2.9 Construct an execution table for the input

```
†12  16  -1  -1†
```

a) to the program `GreatCalculationDone` (Section 21.2.2).

b) to the program `GreatCalculationDone2`.

21.2.10 Instead of creating the structured `GreatCalculationDone6`, use a single GOTO statement to transform `GreatCalculationDone5` (Section 21.2.2) into a program not containing the variable `L1`, but in which the READ statement need not be duplicated.

21.2.11 Demonstrate formally that $g = GCD$ for the WHILE statement W analyzed at the end of Section 21.2.2. In addition to the properties of GCD listed in Section 18.2.4, the following may be helpful:

$GCD(X,X) = X$,
$GCD(X,Y) = GCD(X\text{-}Y,Y)$ for $X>Y$.

21.3 Chapter Summary

It is something of an exaggeration to call programs containing GOTO statements unstructured because GOTO statements can be used in a disciplined manner. However, early programming languages typically contained only IF and GOTO statements to implement all control structures, and undisciplined use of GOTO statements created "spaghetti code" which was very difficult to understand. Disciplined use of GOTO statements fell into familiar execution patterns: selection from alternatives, and bounded or potentially unbounded repetition. As languages matured, special control structures were added to express these patterns and fewer GOTO statements were needed. Programmers sometimes still turn to GOTO statements for efficiency (because otherwise statements would be duplicated) or for separating normal execution from error processing.

CHAPTER 22

REAL NUMBERS

Chapter Preview

Pascal provides an approximation to real numbers in the data type REAL. As with type INTEGER, there may be errors when the size limits of the Pascal machine are exceeded; in addition, a new source of difficulty arises in roundoff errors.

Real numbers may be represented in Pascal with an explicit decimal point or in floating-point (scientific) notation. In floating-point notation, a number is thought of as described by two parts, a real number m called the *mantissa* and an integer e called the *exponent* or *scaling factor*. For some positive integer base b, the number value is

$$m \times b^e.$$

The base is usually a power of two (often sixteen) in computer hardware used for the Pascal machine, but the examples in this chapter will be given in base ten—they are easier to understand and the ideas do not depend on the base. The mantissa is restricted to a range that will make the floating-point representation unique. In the usual base-ten scientific notation this range is [1, 10), that is, all reals r such that $1 \leq r < 10$. Using the letter "E" to separate mantissa and exponent, some examples are:

Decimal Fraction	Scientific Notation
1.0	1.0E0
10.0	1.0E1
3100.0	3.1E3
0.001223	1.223E-3
-923.4	-9.234E2

In most Pascal machines the space available for mantissa and exponent of a floating-point value is limited, which leads to potential arithmetic errors.

22.1 Type REAL

Preview

The REAL data type is convenient for processing which requires fractional values, or where values cover a wide range.

22.1.1 Syntax and Operations

The identifier REAL may be used in a type declaration to specify a value that approximates a real number. REAL constants may be written with an explicit decimal point or in scientific notation (base ten). They may not start or end with a decimal point. The syntax is:

<constant> ::= <sign> <unsigned number> ¦ <unsigned number>
¦ <sign> <constant identifier> ¦ <constant identifier>
¦ <character string>

<unsigned number> ::= <unsigned integer> ¦ <unsigned real>

<sign> ::= + ¦ -

<unsigned real> ::= <unsigned integer> . <digit sequence>
¦ <unsigned integer> . <digit sequence> E <signed integer>
¦ <unsigned integer> E <signed integer>

REAL expressions may use arithmetic and relational operators:

<expression> ::= <simple expression>
¦ <simple expression> <relational operator> <simple expression>

<simple expression> ::= <simple expression> <add operator> <term>
¦ <term> ¦ <sign> <term>

<term> ::= <term> <mult operator> <factor> ¦ <factor>

<factor> ::= <variable access> ¦ <unsigned constant>
¦ <function designator> ¦ (<expression>) ¦ <set constructor>
¦ NOT <factor>

<mult operator> ::= * ¦ /

<add operator> ::= + ¦ -

The binary arithmetic operators produce a REAL result from two REAL operands. INTEGER operands may be mixed with REAL operands, the INGEGER value being treated as if it were a real value instead of an integer. For mixed operations the result returned is REAL. The difference between operating on two INTEGER values and using one or two REAL values with the same operator is a source of confusion. When INTEGER values are used, the result can be in error only if *MAXINT* is exceeded, as described in Section 14.3.3. With mixed or REAL values, the range of values available is much wider, but even when the results stay within range, there can be a loss of precision that never occurs with INTEGER results. (Such errors are treated in Section 22.2 below.) For the division operator, there is a more obvious difference: integer division (÷) and real division (/) are different operations. For example, in the Pascal syntax,

```
3 DIV 2 = 1
```

has value TRUE, but

```
3/2 = 1.5
```

is also TRUE.

REAL is not an ordinal data type; the operations SUCC and PRED are not defined for REAL operands. Thus REAL values may not be used directly as array indices, in FOR statements, etc. Two predefined operations convert REAL values to INTEGER: TRUNC and ROUND. TRUNC returns the INTEGER obtained by discarding the fractional part of the REAL value. ROUND produces the INTEGER closest in value to the REAL. These operations are defined only if the result lies within the *MAXINT* bounds of the INTEGER type. For example:

X	TRUNC (X)	ROUND (X)
5.3	5	5
7.5	7	8
-6.3	-6	-6
-9.7	-9	-10

The predefined operations involving REAL values are:

Operation	Parameter and Result Types
ABS	Real → Real
SQR	Real → Real
EXP	Real (positive) → Real Integer (positive) → Real
LN	Real → Real Integer → Real
SQRT	Real (positive) → Real
ARCTAN	Real (radians) → Real Integer (radians) → Real
SIN	Real (radians) → Real Integer (radians) → Real
COS	Real (radians) → Real Integer (radians) → Real

ABS and SQR have the same meaning as the INTEGER counterparts. EXP computes the exponential function $f(x) = e^x$, and LN computes the natural logarithm (for positive arguments). Since

$$e^{n \ln b} = (e^{\ln b})^n = b^n,$$

a power function like

```
X * X * X * X * X {5th power of X}
```

can be calculated with

```
EXP(5 * LN(X)) {5th power of X}
```

SQRT computes the square root of its positive argument and is otherwise undefined. ARCTAN, SIN, and COS compute these trigonometric functions for arguments expressed in radians.

22.1.2 Real Input/Output

REAL values can be read from and written to TEXT files and enjoy the same kind of automatic conversions that the INTEGER type does. On input, READ statements with REAL variables as actual parameters will take successive values from a TEXT file, and those values may be expressed in the same syntax as REAL constants, either with an explicit decimal point or in scientific notation. On output, values will be printed in a fixed-width field in scientific notation. However, REAL expressions in a WRITE statement can include a formatting directive to specify the number of digits that should appear to the right of a decimal point, in addition to the directive setting the size of the output field. For example, the statement

```
WRITELN(0.015007, 1.23456789E3: 12:2)
```

will add to OUTPUT the string:

```
†1.5007000000000E-02      1234.57†
```

The default field size and precision of the scientific format are defined for each Pascal machine; when the size directives appear, the value is rounded to the number of fractional digits specified, and placed right-justified in the field. Thus in the example above there are 5 blanks to the left of the second number, which occupied only 7 places of the 12-place field when rounded to two fractional digits.

Both the default output format and the one controlled by the programmer can be abused to make it appear that results are more accurate than in fact they are. This topic is discussed in Section 22.2.

22.1.3 Compound Interest and Simulation

The use of type REAL will be illustrated by a simple example and a more complex one.

If an amount of principal P is invested at interest rate r for T years, compounded m times per year, the balance accrued will be

$$P(1 + r/m)^{mT}.$$

Given input containing the principal, interest rate, number of compoundings per year, and years, the following program prints a table of balances at the end of each year.

```
PROGRAM CompoundInterest(INPUT,OUTPUT);
  {Reads principal (real), interest rate (real),
   number of compoundings (integer), and
   years of investment (integer).  Computes
   balance at the end of each year.}
  VAR
    Balance, IRate: REAL;
    Periods, Years, I: INTEGER;
  BEGIN {CompoundInterest}
    READ(Balance,IRate,Periods,Years);
    WRITELN('Principal',Balance:11:2);
    WRITELN('Rate ', 100*IRate:5:2, '%');
    WRITELN('Compounded',Periods:4, ' times');
    WRITELN;
    WRITELN('Year    New Balance');
    FOR I := 1 TO Years
    DO
      BEGIN
        Balance := Balance *
                   EXP(Periods*LN(1+IRate/Periods));
        WRITELN(I:2,'     ',Balance:11:2)
      END
  END.  {CompoundInterest}
```

Execution

```
INPUT :10000.00 0.08 365 20
OUTPUT:Principal   10000.00
       Rate  8.00%
       Compounded 365 times

       Year    New Balance
        1        10832.78
        2        11734.90
        3        12712.16
        4        13770.79
        5        14917.59
        6        16159.89
        7        17505.65
        8        18963.48
        9        20542.71
       10        22253.46
       11        24106.67
       12        26114.22
       13        28288.95
       14        30644.78
       15        33196.80
```

```
16        35961.35
17        38956.13
18        42200.30
19        45714.64
20        49521.64
```

For a more complicated example, Pascal can be used to perform *simulated* experiments on models of systems. Systems are described by a time sequence of events that occur in them. For example, in a banking system, the events might be customers entering a service line and a teller serving them from that line. To simulate such a system, program objects to represent the events are created and stored in a time-ordered sequence. From the sequence a history of the operation of the system can be constructed. Events do not necessarily occur in the order in which they are created. In the bank example, customer X might arrive at time t when there is no line, need 5 minutes of service, and so depart at time t+5. Meanwhile customer Y might arrive at time t+2 needing 1 minute of service. But customer Y cannot be served before the teller has finished with customer X, so Y departs at t+6. The order in which these events are created and occur is:

Creation Order		Occurrence Order	
event	time	event	time
X arrives	t	X arrives	t
X departs	t+5	Y arrives	t+2
Y arrives	t+2	X departs	t+5
Y departs	t+6	Y departs	t+6

The second column of this table describes the system as it is simulated.

Simulations in which all the events are given are less interesting than those in which the events are generated according to some model of the system. For example, the bank customers might arrive at fixed intervals with a random fluctuation superimposed. The service times each customer requires could be imagined as another quantity randomly distributed around some mean. These assumptions about how customers operate are of course oversimplified, but from them the complete system behavior can be derived. The point of simulation is to allow experimentation without incurring the expense of using the real system. If the simulation does not behave as expected, its assumptions can be changed. But no customers get angry when simulated lines are too long. When the events are generated as a part of the simulation, the only input parameter is the length of time for which the simulated system is run.

A program will now be designed to carry out the customer simulation described above.

Design Part 1

```
PROGRAM Simulate(INPUT,OUTPUT);
  {Simulate the arrivals and departures of customers
   in a line until a time read from INPUT has elapsed.}
  TYPE
    EventType = (Arrival, Departure);
    {EventQueue};
  VAR
    Served, Waiting: INTEGER;
    Time, MaxTime: REAL;
```

```
BEGIN {Simulate}
  Time := 0.0;
  READ(MaxTime);
  Served := 0;
  Waiting := 0;
  {create first arrival and departure};
  REPEAT
    {process events in time-ordered sequence}
  UNTIL Time >= MaxTime;
  WRITELN('customers waiting ',Waiting:3);
  WRITELN('customers served ',Served:3)
END.  {Simulate}
```

The objects that represent events are records kept in a data abstraction called a *priority queue*. This structure is like a queue (Section 13.4) except that it is ordered by information in addition to arrival order (here the simulated time). When two events have the same arrival or departure time, they go in the queue in the order they arrived; when an event has an earlier time, it skips ahead to its time-ordered place. The operations on events are like those for queues, but `AddEvent` adds an event to the event sequence so that the priority (time) order is preserved. `PrintEvents` is not really an operation on events, but will be useful for program testing. The essence of the simulation algorithm is the maintenance of a "system time" that ticks off not like a clock, but skips from event to event as they occur.

Design Part 1.1

```
{TYPE EventQueue}
TYPE
  EventQueue = ↑EventElt;
  EventElt = RECORD
               Next: EventQueue;
               Time: REAL;
               Kind: EventType;
               Duration: REAL
             END;
VAR
  Events: EventQueue;
PROCEDURE InitEvent(VAR E: EventQueue);
  {E := empty sequence}
PROCEDURE PrintEvents(VAR E: EventQueue);
  {Display contents of E}
PROCEDURE RemoveEvent(VAR E: EventQueue;
                      VAR Kind: EventType;
                      VAR Length, Time: REAL);
  {Remove the first event on E, return its Kind
   and Length, and update the system time to the
   Time of this event}
PROCEDURE AddEvent(VAR E: EventQueue;
                   Kind: EventType;
                   Start, Length: REAL);
  {Add a new event to E so that E remains in
   increasing time order.  The new event occurs
   at time Start and lasts for time Length.}
```

To start the simulation, the first customer's arrival and departure are scheduled.

Design Part 1.2

```
{create first arrival and departure}
InitEvent(Events);
{generate ArrivalTime and ServiceDelay values};
FreeTime := ArrivalTime + ServiceDelay;
AddEvent(Events,Arrival,ArrivalTime,ServiceDelay);
AddEvent(Events,Departure,FreeTime,0.0);
```

This design part requires the following declarations to be added to Design Part 1:

```
VAR
  ArrivalTime: REAL;  {time of customer arrival}
  ServiceDelay: REAL; {time of customer service}
  FreeTime: REAL;     {time of customer departure}
```

Customers are assumed to arrive at random intervals with a given mean value. The function Random creates a *quasi-random* sequence of integers X_0, X_1, X_2, ... in the range $[0, U)$ by choosing X_0 in this range, and computing

$$X_{i+1} = (MX_i + C)\ MOD\ U,$$

where M and C are suitable constants. These integers are converted to the real range $[0, 1)$ by dividing by U.

Design Part 1.2.1

```
{generate ArrivalTime and ServiceDelay values}
ArrivalTime := ArrivalGap * Random;
ServiceDelay := ServiceTime * Random;
```

The constant declarations:

```
CONST
  ArrivalGap = 10.0;
  ServiceTime = 8.0;
```

should be added to Design Part 1.

Design Part 1.2.1.1

```
FUNCTION Random : REAL;
  CONST
    Modulus = 8192;
    Multiplier = 3125;
  BEGIN {Random}
    Seed := (Multiplier * Seed) MOD Modulus;
    Random := Seed / Modulus
  END;  {Random}
```

Seed must be declared in Design Part 1, and it must be given an initial value, say:

```
Seed := 17;
```

During the simulation, events are removed from the time-ordered queue of events and processed. New events are generated each time an arrival is processed.

Design Part 1.3

```
{process events in time-ordered sequence}
RemoveEvent(Events,KindOfEvent,ServiceDelay,Time);
CASE KindOfEvent OF
  Arrival:
    BEGIN
      WRITELN('customer arrives at ',Time:5:2,
              ' for ',ServiceDelay:5:2);
      Waiting := Waiting + 1;
      {generate next events and add them to the queue}
    END;
  Departure:
    BEGIN
      WRITELN('customer leaves at time ',Time:5:2);
      Waiting := Waiting - 1;
      Served := Served + 1
    END
END
```

The declaration

```
VAR
  KindOfEvent: EventType;
```

must be added as usual.

Each time a customer arrives, the corresponding departure is scheduled. If the teller is busy (`FreeTime > ArrivalTime`), the customer's departure must be delayed until after the teller is free. However, if `FreeTime <= ArrivalTime`, the teller is idle and can serve the customer immediately.

Design Part 1.3.1

```
{generate next events and add them to the queue}
ArrivalTime := Time + ArrivalGap * Random;
ServiceDelay := ServiceTime * Random;
IF FreeTime > ArrivalTime
THEN {teller busy, schedule delayed service}
  FreeTime := FreeTime + ServiceDelay
ELSE {teller free, schedule immediate service}
  FreeTime := ArrivalTime + ServiceDelay;
AddEvent(Events,Arrival,ArrivalTime,ServiceDelay);
AddEvent(Events,Departure,FreeTime,0.0)
```

Piecing these design parts together gives:

Design Part 1

```
PROGRAM Simulate(INPUT,OUTPUT);
  {Simulate the arrivals and departures of customers
   in a line until a time read from INPUT has elapsed.}
  CONST
    ArrivalGap = 10.0;
    ServiceTime = 8.0;
  TYPE
    EventType = (Arrival, Departure);
    {EventQueue};
  VAR
    Served, Waiting: INTEGER;
    Time, MaxTime: REAL;
    ArrivalTime: REAL;  {time of customer arrival}
    ServiceDelay: REAL; {time of customer service}
    FreeTime: REAL;     {time of customer departure}
    Seed: INTEGER;
    KindOfEvent: EventType;
  FUNCTION Random : REAL;
    CONST
      Modulus = 8192;
      Multiplier = 3125;
    BEGIN {Random}
      Seed := (Multiplier * Seed) MOD Modulus;
      Random := Seed / Modulus
    END;  {Random}
  BEGIN {Simulate}
    Time := 0.0;
    READ(MaxTime);
    Served := 0;
    Waiting := 0;
    Seed := 17;
    {create first arrival and departure}
    InitEvent(Events);
    {generate ArrivalTime and ServiceDelay values}
    ArrivalTime := ArrivalGap * Random;
    ServiceDelay := ServiceTime * Random;
    FreeTime := ArrivalTime + ServiceDelay;
    AddEvent(Events,Arrival,ArrivalTime,ServiceDelay);
    AddEvent(Events,Departure,FreeTime,0.0);
    REPEAT
      {process events in time-ordered sequence}
      RemoveEvent(Events,KindOfEvent,ServiceDelay,Time);
      CASE KindOfEvent OF
        Arrival:
          BEGIN
            WRITELN('customer arrives at ',Time:5:2,
                    ' for ',ServiceDelay:5:2);
            Waiting := Waiting + 1;
            {generate next events and add them to the queue}
            ArrivalTime := Time + ArrivalGap * Random;
            ServiceDelay := ServiceTime * Random;
            IF FreeTime > ArrivalTime
            THEN {teller busy, schedule delayed service}
```

```
            FreeTime := FreeTime + ServiceDelay
          ELSE {teller free, schedule immediate service}
            FreeTime := ArrivalTime + ServiceDelay;
          AddEvent(Events,Arrival,ArrivalTime,ServiceDelay);
          AddEvent(Events,Departure,FreeTime,0.0)
        END;
      Departure:
        BEGIN
          WRITELN('customer leaves at time ',Time:5:2);
          Waiting := Waiting - 1;
          Served := Served + 1
        END
    END
  UNTIL Time >= MaxTime;
  WRITELN('customers waiting ',Waiting:3);
  WRITELN('customers served ',Served:3)
END.  {Simulate}
```

The procedures that implement the operation for a priority event queue construct the queue as a singly-linked list in time order by arrival.

Design Part 1.1.1

```
PROCEDURE InitEvent(VAR E: EventQueue);
  {E := empty sequence}
  BEGIN {InitEvent}
    E := NIL
  END;  {InitEvent}
```

Design Part 1.1.2

```
PROCEDURE PrintEvents(VAR E: EventQueue);
  {Display contents of E}
  VAR
    TPtr: EventQueue;
  BEGIN {PrintEvents}
    TPtr := E;
    WRITELN;WRITELN;
    WHILE TPtr <> NIL
    DO
      BEGIN
        WRITE(TPtr↑.Time:8:3);
        IF TPtr↑.Kind = Arrival
        THEN
          WRITELN(' customer arrives for',
                  TPtr↑.Duration:5:2)
        ELSE
          WRITELN(' teller free');
        TPtr := TPtr↑.Next
      END;
    WRITELN;WRITELN
  END;  {PrintEvents}
```

Design Part 1.1.3

```
PROCEDURE RemoveEvent(VAR E: EventQueue;
                      VAR Kind: EventType;
                      VAR Length, Time: REAL);
  {Remove the first event on E, return its Kind
   and Length, and update the system time to the
   Time of this event}
  BEGIN {RemoveEvent}
    IF E <> NIL
    THEN
      BEGIN
        Time := E↑.Time;
        Kind := E↑.Kind;
        IF Kind = Arrival
        THEN
          Length := E↑.Duration;
        E := E↑.Next
      END
  END;  {RemoveEvent}
```

Design Part 1.1.4

```
PROCEDURE AddEvent(VAR E: EventQueue;
                   Kind: EventType;
                   Start, Length: REAL);
  {Add a new event to E so that E remains in
   increasing time order.  The new event occurs
   at time Start and lasts for time Length.}
  VAR
    EPtr, LPtr, CPtr: EventQueue;
    Found: BOOLEAN;
  BEGIN {AddEvent}
    {create new event referenced by EPtr};
    IF E = NIL
      THEN {empty sequence}
        {insert first event}
      ELSE {sequence with one or more events}
        BEGIN
          {LPtr=previous event; CPtr=following event}
          IF LPtr = NIL
          THEN {insert as new first event}
            E := EPtr
          ELSE {insert as future event}
            LPtr↑.Next := EPtr;
          EPtr↑.Next := CPtr
        END
  END;  {AddEvent}
```

Although the records created are all the same size, only those representing customer arrivals have values in the Duration field.

Design Part 1.1.4.1

```
{create new event referenced by EPtr}
NEW(EPtr);
IF Kind = Arrival
THEN
  EPtr↑.Duration := Length;
EPtr↑.Kind := Kind;
EPtr↑.Time := Start;
```

Design Part 1.1.4.2

```
{insert first event}
BEGIN
  E := EPtr;
  E↑.Next := NIL
END
```

LPtr and CPtr are calculated so that CPtr points to each event in turn, and LPtr to the previous event. When the correct position is found, the event is linked into the priority queue.

Design Part 1.1.4.3

```
{LPtr=previous event; CPtr=following event}
LPtr := NIL;
CPtr := E;
Found := FALSE;
WHILE (CPtr <> NIL) AND NOT Found
DO
  BEGIN
    IF CPtr↑.Time > Start
    THEN
      Found := TRUE
    ELSE
      BEGIN
        LPtr := CPtr;
        CPtr := CPtr↑.Next
      END
  END;
```

A simulation for 25 time units is shown below; in this time 7 customers arrive and 6 leave.

Execution

```
INPUT :25.0
OUTPUT:customer arrives at  4.85 for  4.63
       customer arrives at  5.59 for  6.31
       customer arrives at  7.61 for  4.56
       customer leaves at time  9.48
       customer arrives at 13.17 for  2.86
       customer leaves at time 15.80
       customer arrives at 16.42 for  0.73
       customer arrives at 18.37 for  1.66
       customer leaves at time 20.35
       customer arrives at 21.91 for  5.15
       customer leaves at time 23.21
       customer leaves at time 23.94
       customer leaves at time 25.60
       customers waiting   1
       customers served    6
```

22.1.4 Exercises

22.1.1 In the following Pascal fragment which statements are syntactically correct? What is the type of each expression in each WRITE statement?

```
VAR
  R1, R2: REAL;
  I1, I2: INTEGER;
BEGIN
  ...
  I1 := R1;
  R2 := I2;
  WRITELN(I1 - TRUNC(R1*I2));
  WRITELN(I1 < R1);
  WRITELN(I1 + R1 < R1);
  WRITELN(I1 + R1 - I2);
  WRITELN(I1 / R1 + I2)
END
```

22.1.2 Explain the rationale that lies behind the Pascal restriction that the controlled variable of a FOR statement may not be REAL.

22.1.3 Investigate what your Pascal machine does for REAL input/output, in particular:

a) What is the default format and field width for REAL output?

b) Does the problem of end of file with INTEGER input (discussed in Section 14.3.2) exist for REAL input as well?

c) What happens on output if the field width is inadequate?

d) Are leading or trailing decimal points allowed on input?

e) What happens on output if the number of digits after the decimal point is specified to be more than the actual precision available?

f) Is it possible to read in a REAL value, then write it, so that the result appears different than what was in the input?

22.1.4 Write a procedure taking one REAL parameter that prints this value in "dollars and cents format"; that is, with a leading $ at the left (no spaces offset), and the number divided into groups of three digits separated by commas, and rounded to two digits following the decimal point. For example, 1.234566666E5 should appear as $123,456.67.

22.1.5 Design a module that implements the data abstraction RealI in which a real number is represented using type INTEGER.

a) Represent the real number by two INTEGER values, one for its mantissa and one for its exponent. Implement the operations of input conversion (from two INTEGER parameters), multiplication, addition, and output conversion (print directly in scientific notation). Arrange to give correct results so long as it is possible to keep both values within the bounds of *MAXINT*, represented in the program by a constant declaration.

b) Repeat a), but for a representation in which the two values are the part of the real number before and after the decimal point.

22.1.6 Change the representation of type Rational in Section 18.2.1 from two INTEGER values to a single REAL value, and make the necessary changes in the operation implementations. Explain the changes that will be required in HarmonicSeries (Section 18.2.3) because of the new representation.

22.1.7 Design a module to implement type Complex. A complex number $z = a + ib$ where $i = \sqrt{(-1)}$ should be represented by REAL components a and b. Implement the operations of addition and multiplication for Complex. Devise an appropriate input/output format and write a program using your module which reads two complex numbers and prints their sum and product.

22.1.8 Check the formula for compound interest (Section 22.1.3) by solving the same problem using the brute-force approach. That is, at each compounding time, calculate the interest and add it to the principal. Compare the brute-force and formula results for the case of $1 invested at 5% for 1 year, compounded ***hourly***.

22.1.9 Using the formula for compound interest in Section 22.1.3, write a program that calculates the necessary interest rate to the nearest ½%, compounded daily, to realize a given balance starting from a given principal over a given number of years. That is, take the starting principal, number of years, and ending balance as input, and find the needed rate.

a) Use binary search to solve the problem by trial and error.

b) Solve the formula for r and evaluate it.

c) Compare the speed and ease of obtaining the programs in a) and b).

22.1.10 Modify the simulation program of Section 22.1.3 so that

a) in the output the customers are identified by a number assigned on arrival: 1,2,3,... Explain why it is not necessary to store this number in the event records, but possible to instead calculate it when the printing is done. (But do the implementation any way you like.)

b) the simulation completes a number of customers taken from the input, instead of a maximum time.

c) at the end of the simulation, information is given about how well the system worked, in particular the maximum length of the line waiting for the teller, the longest wait in line by any customer, the number of customers served without waiting, and the total time when the teller was idle. If necessary, adjust the parameters of the simulation so that interesting values arise for each quantity.

d) in addition to the assumptions about customer arrival and service needs already implemented, there are two "bursts" of activity, one at the middle of the simulated time, and the other at the end. At a burst time, N customers arrive all together, where N is half the number whose service has been completed up to that time.

22.2 Roundoff Errors

Preview

The REAL data type is usually implemented using special hardware in a Pascal machine, hardware in which both mantissa and exponent are limited in size, and so subject to overflow. Conversion between number bases for input/output can introduce additional errors.

Scientific calculations usually make use of real numbers, and the speed with which the necessary arithmetic can be executed may be the limiting factor in deciding whether or not a problem can be solved in a reasonable time. (Speeds are measured in "megaflops"—millions of floating-point operations per second; 10 megaflops is a fast machine.) However, the payment for speed is a potential for error: the way REAL quantities are stored for fast arithmetic limits both mantissa and exponent in size. This means that the range of real values is limited, and so is the precision. When overflow occurs in the range, the situation is much like that for type INTEGER. The range of REAL values is greater, but that range can be exceeded, particularly by repeated multiplication or division. The difficulty with precision is more subtle, and falls under the heading of *roundoff* error.

Because the mantissa is limited, only a certain number of digits (thinking in base ten) can be stored. If a value requires more digits than there are available, it cannot be exactly represented and must be "rounded off" to the nearest value that is available. The most important cause of roundoff error is the natural loss of precision in intermediate computations. For example, consider the sum:

$$0.37\times10^3 + 0.25\times10^2 = 0.37\times10^3 + 0.025\times10^3 = (0.37+0.025)\times10^3 = 0.395\times10^3$$

If the calculation must be carried out in base ten with two-digit accuracy, the final value cannot be represented, so it is rounded to 0.40×10^3. In the example, the two-digit precision is retained despite roundoff; however, in a calculation like:

$$0.37\times10^3 + 0.25\times10^2 - 0.15\times10^2$$

the rounded result should be 0.38×10^3, but rounding each calculation from left to right yields 0.39×10^3, a real loss of precision.

Roundoff errors also occur in converting from one number base to another. This is a source of trouble in Pascal because REAL values are expressed in base ten in program constants and input values, but must often be converted to a base that is a power of two for fast use in the Pascal machine. A number representable with limited precision in one base may fail to have a limited precision representation in another base. For example, the number 0.37×10^1 (base ten) has no exact counterpart in base two. Suppose it were to be converted to base two with a mantissa limited to six binary places. The following table will be helpful:

Equivalent Values

base two	base ten
.1	.5
.01	.25
.001	.125
.0001	.0625
.00001	.03125
.000001	.015625

Conversion of the fractional part proceeds by repeatedly finding the largest value in the base-ten column and subtracting it from the remainder to be converted, at the same time adding the base-two equivalent to the result.

$$\begin{aligned}
3.7_{10} &= 11_2 + 0.7_{10} \\
&= (11_2 + 0.1_2) + (0.7_{10} - 0.5_{10}) \\
&= 11.1_2 + 0.2_{10} \\
&= 11.1_2 + 0.001_2 + (0.2_{10} - 0.125_{10}) \\
&= 11.101_2 + 0.075_{10} \\
&= 11.101_2 + 0.0001_2 + (0.075_{10} - 0.0625_{10}) \\
&= 11.1011_2 + 0.0125_{10}
\end{aligned}$$

Rounded to six "bigets" (usually called *bits*) the result is $11.1011 = 0.111011\times2^2$. Errors of this kind are called *representational errors*.

The approximate nature of REAL computations implies that programs should not test REAL values for equality. The following Pascal fragment may never terminate because successive additions of 1/10 to 0.0 may never yield the exact value 1.0.

```
RValue := 0.0;
WHILE RValue <> 1.0 DO
  BEGIN
    ... {loop body}
    RValue := RValue +  1 / 10
  END
```

It is much better to control the loop with the test:

```
RValue < 1.0
```

The usual course of scientific computation is to read input data (and express program constants) in base ten, convert to the internal base of the Pascal machine, carry out many arithmetic operations, then convert the result back to base ten for output. Each conversion and

arithmetic operation carries the potential for roundoff error. Base-conversion roundoff occurs only once for each value, so its worst case can be found quite easily—no more than the final digit is usually affected. (However, the appearence of .99999999 for 1.0000000 is annoying.) Arithmetic operations can be performed repeatedly, so there is the potential for seriously compromising the validity of the calculation through operation roundoff. The roundoff in each operation is limited, but the cumulative effect of many operations on the same variable can become large—in fact, larger than the final value of the variable itself. Such accumulated roundoffs can not only lead to large final errors in data, but can also affect the conditional and iteration tests of a program, leading to unexpected execution sequences.

In analyzing the roundoff errors in a computation, the *relative errors* (that is, the ratio of error to exact value) in each variable are the most interesting. Consider evaluating the polynomial

$$0.16x^3 + 0.86x^2 - 2.9x + 7.4$$

at x=2.3 in base-ten arithmetic rounded to two significant decimal digits, using Horner's rule:

$$(((((0.16\times2.3) + 0.86)\times2.3) - 2.9)\times2.3) + 7.4$$

The operations and roundoff errors at each step are shown below.

Arithmetic step	Rounded operation	Actual value	Computed value	Local error
1	0.16×2.3	0.368	0.37	0.002
2	0.37+0.86	1.23	1.2	-0.03
3	1.2×2.3	2.76	2.8	0.04
4	2.8-2.9	-0.1	-0.10	0.00
5	-0.10×2.3	-0.23	-0.23	0.00
6	-0.23+7.4	7.17	7.2	0.03

The Actual value column is the result of a perfect calculation, but using the rounded value (Computed value column) from the previous step. The last column (Local error) is the difference between the actual and rounded result for each step. However, the complete story is not told in this table, because it takes no account of the accumulation of errors throughout the whole computation. Imagine that the "perfect" results of each operation (signified by a symbol within a circle) were available. Then the following table could be constructed:

Arithmetic step	Perfect operation	Perfect value	Total error	Relative error
1	0.16 ⊗ 2.3	0.368	0.002	0.00540 ...
2	0.368 ⊕ 0.86	1.228	-0.028	-0.0233 ...
3	1.228 ⊗ 2.3	2.8244	-0.0244	-0.00871 ...
4	2.8244 ⊖ 2.9	-0.0756	-0.0244	0.244
5	-0.0756 ⊗ 2.3	-0.17388	-0.5612	0.286
6	-0.17388 ⊕ 7.4	7.22612	-0.02612	-0.00362 ...

The last two columns are the difference between the composite rounded result (Computed value in the previous table) and the perfect value, expressed as a magnitude (Total error) and as a fraction of the perfect value (Relative error).

Errors accumulate in complex and unexpected ways even in this simple calculation. In step 4, the relative error jumps from less than 1% to more than 24%. The local error in step 4 is 0, but a previously accumulated error is amplified because the difference of two nearly equal numbers is computed. Steps 4 and 5 have large relative errors, but the final result of step 6 does not. This is the happy result of the dominance of the perfect value 7.4 in the computation.

22.2.1 Roundoff Notation

In order to study the effects of roundoff more systematically, the following notation is adopted for an operation at step i, using the results of previous steps j and k. The rounded operation is represented by +, while $\oplus$ is the corresponding perfect operation.

Step	Rounded operation	Computed value	Local error	Perfect operation	Perfect value	Total error
i	$q_j + q_k$	q_i	w_i	$r_j \oplus r_k$	r_i	e_i

The definitions of these quantities are:

(1) $q_i = q_j + q_k$
(2) $r_i = r_j \oplus r_k$
(3) $w_i = q_j + q_k - (q_j \oplus q_k)$
(4) $e_i = q_i - r_i$

This notation can be applied to the polynomial evaluation above as follows. Negative indices will identify initial data (for which q_i and r_i are the same), and positive indices will be used for intermediate and final values. The initial data consist of the values:

0.16, 0.86, -2.9, 7.4, 2.3

which are arbitrarily called q_{-1} through q_{-5}.

Initial Data					
i	-1	-2	-3	-4	-5
q_i	0.16	0.86	-2.9	7.4	2.3

The six values computed in order are:

Computed Data						
step i	1	2	3	4	5	6
value q_i	0.37	1.2	2.8	-0.10	-0.23	7.2

Furthermore, the six steps had operation and operand indices given by the table below.

Operation and Operand Indices						
step i	1	2	3	4	5	6
operand j	-1	1	2	3	4	5
operation	×	+	×	+	×	+
operand k	-5	-2	-5	-3	-5	-4

These tables conveniently record all the information needed to analyze roundoff error.

22.2.2 An Analysis of Roundoff Error

The definitions (1) - (4) above can be solved for the accumulated error e_i, which will then be expressed in terms of local errors and intermediate values. To be useful, any error analysis must be given in terms of *observable* values. If the perfect calculation were available, the error could be calculated directly, but in that case there would be no reason to do so—the perfect value could be used and the error made zero! Since perfect values are not available, the quantities that should occur in formulas are computed values q_i. The local error w_i is not observable, but since it results from a single roundoff its maximum size can be found. Starting with definition (4):

$$
\begin{aligned}
e_i &= q_i - r_i \\
&= q_j + q_k - (r_j \oplus r_k) \quad \text{(using (1) and (2))} \\
&= q_j \oplus q_k + w_i - (r_j \oplus r_k) \quad \text{(using (3))} \\
&= q_j \oplus q_k - ((q_j - e_j) \oplus (q_k - e_k)) + w_i \quad \text{(using (4))}.
\end{aligned}
$$

That is, the accumulated error e_i in a new computed value q_i is expressed in terms of the old observable values q_j and q_k used in computing it, their errors e_j and e_k, and the local error w_i. In the example above, substituting the computed values q_i yields:

$$
\begin{aligned}
e_1 &= (0.16 \otimes 2.3) - (0.16 \otimes 2.3) + w_1 = w_1 \\
e_2 &= (0.37 \oplus 0.86) - ((0.37 - e_1) \oplus 0.86) + w_2 = e_1 + w_2 \\
e_3 &= (1.2 \otimes 2.3) - ((1.2 - e_2) \otimes 2.3) + w_3 = 2.3e_2 + w_3 \\
e_4 &= (2.8 \ominus 2.9) - ((2.8 - e_3) \ominus 2.9) + w_4 = e_3 + w_4 \\
e_5 &= (-0.10 \otimes 2.3) - ((-0.10 - e_4) \otimes 2.3) + w_5 = 2.3e_4 + w_5 \\
e_6 &= (-0.23 \oplus 7.4) - ((-0.23 - e_5) \oplus 7.4) + w_6 = e_5 + w_6
\end{aligned}
$$

This system of equations can be solved for e_i by substitution:

$$
\begin{aligned}
e_1 &= w_1 \\
e_2 &= w_1 + w_2 \\
e_3 &= 2.3w_1 + 2.3w_2 + w_3 \\
e_4 &= 2.3w_1 + 2.3w_2 + w_3 + w_4 \\
e_5 &= 2.3^2w_1 + 2.3^2w_2 + 2.3w_3 + 2.3w_4 + w_5 \\
e_6 &= 2.3^2w_1 + 2.3^2w_2 + 2.3w_3 + 2.3w_4 + w_5 + w_6
\end{aligned}
$$

Equations expressing the accumulated errors in terms of the local errors may be found in this way for any calculation, but in general the form is more complex, involving ratios of multinomials in the local errors and computed values.

To obtain the value e_6 requires only estimating the local errors. Each is the result of a roundoff to two decimal digits. The worst case occurs when all of the local errors take the largest possible value, and all are in the same direction. For example, if every one of the computed values had a third significant figure of 5 it would be rounded up, and no local errors would cancel. The largest possible rounding error depends on the exponent of the value being rounded—it is the value of a 5 in the third significant figure. Thus for the polynomial evaluation:

Local Error Magnitudes						
i	1	2	3	4	5	6
q_i	0.37	1.2	2.8	-0.10	-0.23	7.2
maximum w_i	0.005	0.05	0.05	0.005	0.005	0.05

The magnitude of the final error is therefore:

$$|e_6| \leq 2.3^2(0.055) + 2.3(0.055) + 0.055 = 0.472...$$

This maximum possible error is much larger than that observed.

22.2.3 Exercises

22.2.1 Find a number that can be exactly represented in base ten with two-digit precision, but when converted into base two with four-bit precision, then converted back to base ten with two-digit precision, is not the original value. Explain why this sort of thing does not happen in Pascal input-output conversion.

22.2.2 Using real values with rounding to two decimal digits, construct examples to demonstrate that the usual associative and distributive laws of arithmetic do not hold. What lesson can be drawn from these examples by the Pascal scientific programmer?

22.2.3 Calculate the area of a rectangle of width 5 and height 2 by dividing the width into 2^k pieces, $k = 0, 1, 2, \ldots$, and adding the area of component rectangles of height 2 based on each piece. Continue until a large enough value of k shows the effect of substantial roundoff error. What lesson can be drawn from this example by the Pascal programmer doing numerical integration?

22.2.4 Consider the solutions

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

to the quadratic equation $ax^2 + bx + c = 0$.

a) The Pascal fragment:

```
Rt := SQRT(B*B - 4*A*C);
Den := 2*A;
X1 := (Rt - B)/Den;
X2 := -(B + Rt)/Den
```

can be used to obtain these roots. If b is a large negative number and a and c are small positive numbers, problems can arise. Find problem values of a, b, and c.

b) The value of x_2 can be computed from x_1 as follows:

$$\begin{aligned} x_1 \times x_2 &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \times \frac{-b - \sqrt{b^2 - 4ac}}{2a} \\ &= \frac{(-b + \sqrt{b^2 - 4ac})(-b - \sqrt{b^2 - 4ac})}{4a^2} \\ &= \frac{b^2 - b^2 + 4ac}{4a^2} = \frac{4ac}{4a^2} = \frac{c}{a} \end{aligned}$$

Thus, after computing x_1 with the Pascal fragment

```
Rt := SQRT(B*B - 4*A*C);
Den := 2*A;
IF B < 0
THEN
  X1 := (Rt - B)/Den
ELSE
  X1 := -(B + Rt)/Den
```

x_2 can be computed with:

```
X2 := C/(X1*A)
```

Check to see if the problem that arose in (a) persists in this method, and explain.

22.2.5 Consider polynomial evaluation by a different method than Horner's rule. Compute the needed powers of the variable x by repeated multiplication, then multiply each by its coefficient, and finally add the terms beginning with the highest-power term and continuing down to the constant term.

a) Obtain the formula for the final accumulated error when the polynomial

$$0.16x^3 + 0.86x^2 - 2.9x + 7.4$$

from Section 22.2 is evaluated in this way.

b) Compare the results with the Horner-rule analysis of Section 22.2.2.

22.2.6 Prove that when the polynomial

$$C_n x^n + C_{n-1} x^{n-1} + \ldots + C_1 x + C_0$$

is evaluated using Horner's rule the accumulated roundoff error is

$$x^{n-1}(w_1+w_2) + x^{n-2}(w_3+w_4) + \ldots + (w_{2n-1}+w_{2n})$$

where w_1, w_2, ... w_{2n} are the local errors in the 2n multiply/add operations required. Note that the result is independent of the intermediate computed values and the coefficient values!

22.3 Chapter Summary

The data type REAL comprises a wider class of values than INTEGER, both in magnitude and by the inclusion of a fractional part. However, REAL values only approximate real numbers, and REAL operations only approximate real arithmetic. It is difficult to estimate the size of the errors that occur when the practical limitations of the Pascal machine come into play, but without an analysis of possible errors the results of calculations can be arbitrarily wrong.

CHAPTER 23

PLANNING FOR CHANGE

Chapter Preview

Programs change because the problems they must solve change, but not radically enough to justify writing a new program. By anticipating changes, and by the use of techniques to localize and protect information and operations, the programmer can make it significantly easier to modify a program.

Programming with abstract data types makes changing programs easier because implementation-dependent information about objects is centralized in type definitions and operation implementations rather than scattered through several users' programs. This chapter examines the use of constant and type declarations in making programs easier to modify. New language features, procedure/function parameters and variant records, are introduced to aid in writing software fitting many applications without requiring difficult alterations.

23.1 CONST and TYPE Declarations

Preview

Centralizing information in CONST and TYPE declarations is a way to allow easy program change, particularly when these declarations are used to control module implementations. However, it is all too easy to build unwarranted assumptions into a program so that extensive, difficult changes are required.

In Section 19.2.2, a Stack was represented using an array and an index to its top element. The declarations hiding the size and representation were:

```
CONST
  Depth = 20;
TYPE
  EltType = CHAR;
  Stack = RECORD
            Val: ARRAY [1..Depth] OF EltType;
            StackTop: 0..Depth
          END;
```

Using a constant declaration for Depth keeps the INTEGER 20 out of the further declarations and out of the Push and Full operations. A single change can therefore expand the Stack size, and it is very likely that the changed module will perform correctly if the original one did. Similarly, by making EltType a TYPE instead of spreading CHAR throughout the module, it is easy to create (say) a Stack of SET values in place of characters. The changes that are thus enabled are far-reaching. For example, to stack up to 2500 personnel records, the only changes required are:

```
CONST
  Depth = 2500;
TYPE
  Month = (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);
  Date = RECORD
           Mo: Month;
           Day: 1 .. 31;
           Year: INTEGER
         END;
  PRecord = RECORD
              ...;
              BirthDay: Date;
              ...
            END;
  EltType = PRecord;
```

Not every possible change can be made in this way. For example, the array subscripts cannot be changed to CHAR values by setting

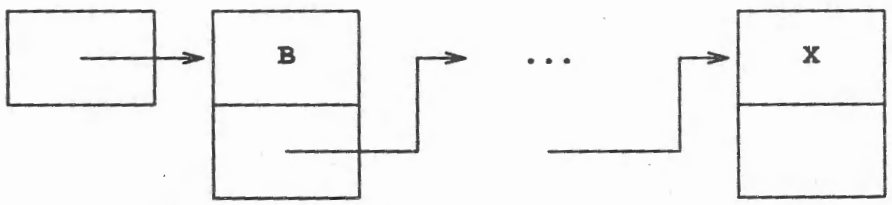
```
Depth = 'Z';
```

because Depth is used in ranges beginning with an INTEGER constant, and the operations use INTEGER operations to manipulate the subscripts. Nor can a stack of files be created by

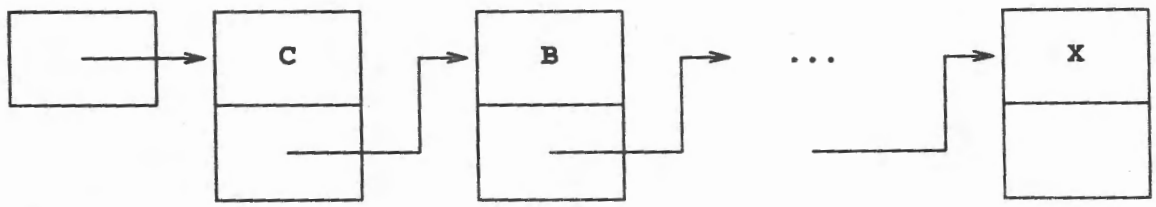
```
EltType = TEXT;
```

because assignments are used to move the Stack elements, and files may not be assigned. It is some consolation that an attempt to make these changes would result in syntactically incorrect Pascal programs, so at least the programmer could not be misled into believing the change was accomplished when it was not. When changes must be made in information that is distributed, it is all too common to think a change is accomplished, and receive no syntactic clues that something has been missed.

The Stack representation as an array and a pointer is fundamental to the module implementation, yet it could require changing. Suppose that in an application where use of computer memory is very expensive, stacks are required to grow large and shrink again. It might be discovered that allocating an array large enough for the maximum size is too wasteful. Instead, the representation should be a singly linked list of the stacked values in order from top to bottom. Then, for example, a Stack of CHAR with **B** on the top and **X** on the bottom would appear as:



and pushing **C** would alter the structure to:



The text of a complete program utilizing Stack appears below. The module application Reverse is unchanged because the module hides the form of Stack, but the module implementation itself had to be completely rewritten.

```
PROGRAM Reverse(INPUT,OUTPUT);
  {reverse the characters in INPUT and write them
   to OUTPUT}
  TYPE
    EltType = CHAR;
    Stack = ↑StackRep;
    StackRep = RECORD
                 Val: EltType;
                 Next: Stack
               END;
  VAR
    S: Stack;
    Elt: EltType;
  PROCEDURE NewStack(VAR S: Stack);
    {S := <>}
    BEGIN {InitStack}
      S := NIL
    END;  {InitStack}
  PROCEDURE Push(VAR S: Stack; E: EltType);
    {S := S & <E>}
    VAR
      Temp: Stack;
    BEGIN {Push}
      Temp := S;
      NEW(S);
      S↑.Val := E;
      S↑.Next := Temp
    END;  {Push}
  PROCEDURE Pop(VAR S: Stack);
    {there are Stack X, EltType E such that
        S = X & <E> --> S := X }
    VAR
      Temp: Stack;
    BEGIN {Pop}
      IF S <> NIL
      THEN {not already empty}
        BEGIN
          Temp := S;
          S := S↑.Next;
          DISPOSE(Temp)
        END
    END;  {Pop}
  FUNCTION Top(VAR S: Stack): EltType;
    {there are Stack X, EltType E such that
        S = X & <E> --> Top := E }
    BEGIN {Top}
      IF S = NIL
      THEN
        WRITELN('** READING EMPTY STACK **')
      ELSE
        Top := S↑.Val
    END;  {Top}
```

```
FUNCTION Empty(VAR S: Stack): BOOLEAN;
  {Empty := (S = <>)}
  BEGIN {Empty}
    Empty := S = NIL
  END;  {Empty}

FUNCTION Full(VAR S: Stack): BOOLEAN;
  {Full := (Length(S) = Depth)}
  BEGIN {Full}
    Full := FALSE
  END;  {Full}
BEGIN {Reverse}
  NewStack(S);
  WHILE NOT EOF AND NOT Full(S)
  DO
    BEGIN
      READ(Elt);
      Push(S,Elt)
    END;
  WHILE NOT Empty(S)
  DO
    BEGIN
      WRITE(Top(S));
      Pop(S)
    END;
  WRITELN
END.  {Reverse}
```

23.1.1 Exercises

23.1.1 Describe in detail the changes that would be required to use the `Stack` module of Section 19.2.2 for a `Stack` of

a) `REAL` values.

b) `Month` values.

c) `SET OF CHAR` values.

d) `TEXT` values.

e) `Stack` of `CHAR` values.

23.2 PROCEDURE and FUNCTION Parameters

Preview

Parameters permit the actions of a procedure to be tailored to different applications on different calls. A surprising amount of power is added to this mechanism by allowing the parameter to itself be a procedure or function.

Procedures and functions can be passed as parameters. In order to indicate that a formal parameter is to be a procedure or a function, the name of the formal is prefixed with either the reserved word `PROCEDURE` or `FUNCTION`. The number and type for each parameter and the result type of a function parameter must be specified. The procedure `Invoke` shown below has a procedure parameter `ProcQ` and a function parameter `ProcR` that itself has no parameters but returns an `INTEGER`. The body of `Invoke` contains procedure statements that actually call the

parameter routines:

```
PROCEDURE Invoke(PROCEDURE ProcQ(Z: INTEGER);
                 FUNCTION ProcR: INTEGER);
VAR
  IVar: INTEGER;
BEGIN {Invoke}
  ProcQ(IVar);
  IVar := ProcR + IVar
END {Invoke}
```

If Invoke is called in a fragment like the following:

```
PROCEDURE Invoke(PROCEDURE ProcQ(Z: INTEGER);
                 FUNCTION ProcR: INTEGER);
  ...;
PROCEDURE OneInt(FParm: INTEGER);
  ...;
FUNCTION FuncF: INTEGER:
  ...;
BEGIN
  ...
  Invoke(OneInt, FuncF);
  ...
END
```

then OneInt and FuncF are not called as a part of the call on Invoke. Instead, these routines are made available so that they are called when the statements within Invoke use the parameter names. The procedure statement ProcQ(IVal) in Invoke actually calls OneInt and the function call ProcR actually calls FuncF.

23.2.1 Syntax for Parameters

The following syntax rules describe formal procedure and function parameters.

<formal parameter> ::= <value parameter>
¦ <variable parameter>
¦ <procedural parameter>
¦ <functional parameter>

<procedural parameter> ::= <procedure heading>

<functional parameter> ::= <function heading>

<actual parameter> ::= <expression> ¦ <procedure identifier>
¦ <function identifier>

The procedure or function passed as actual parameter must match the formal parameter exactly, as illustrated by the following:

```
PROGRAM Main (OUTPUT);
  PROCEDURE TakeProc(PROCEDURE Proc(X: INTEGER));
    BEGIN {TakeProc}
      Proc(7)
    END;  {TakeProc}
  PROCEDURE Proc1(F1, F2: INTEGER);
    BEGIN {Proc1}
      WRITELN(F1+F2)
    END;  {Proc1}
  PROCEDURE Proc2(VAR F1: INTEGER);
    BEGIN {Proc2}
      WRITELN(F1);
    END;  {Proc2}
  PROCEDURE Proc3(F1: INTEGER);
    BEGIN {Proc3}
      WRITELN(F1)
    END;  {Proc3}
  BEGIN {Main}
    TakeProc(Proc1); {wrong number of parameters}
    TakeProc(Proc2); {wrong kind of parameters}
    TakeProc(Proc3)  {OK}
  END.  {Main}
```

The first procedure statement is incorrect because `TakeProc` requires a parameter that is a procedure with a single parameter while `Proc1` has two parameters. The second procedure statement is also incorrect because although `Proc2` has one parameter, it is a variable one and `TakeProc` requires it to be a value one.

23.2.2 Point of Evaluation

A procedure cannot appear as an actual parameter in a call unless the corresponding formal parameter is declared `PROCEDURE`. The natural way to think about such a parameter is the correct way: that its name is being passed for later invocation. However, function names may appear as parameters in two ways: the function may be an expression being passed to a value parameter, or it may be a name passed to a `FUNCTION` parameter. There is a crucial difference between these two meanings, but their syntactic form can be the same. When a function identifier is bound to a value parameter in a procedure call, the function is called, the value returned by the function is bound to the value parameter, and the procedure is invoked. When a function identifier is bound to a `FUNCTION` parameter in a procedure call, no call of the parameter function takes place. Instead, the parameter name is supplied so that the procedure may later call it. The program `Passing` illustrates the difference between these two types of binding.

```
PROGRAM Passing(OUTPUT);
  PROCEDURE TakeInteger(IVar: INTEGER);
    {Parameter is value}
    BEGIN {TakeInteger}
      WRITELN('enter TakeInteger');
      WRITELN(IVar);
      WRITELN('leave TakeInteger')
    END; {TakeInteger}
  PROCEDURE TakeFunction(FUNCTION IFunc: INTEGER);
    {Parameter is FUNCTION}
    BEGIN {TakeFunction}
      WRITELN('enter TakeFunction');
      WRITELN(IFunc);
      WRITELN('leave TakeFunction')
    END; {TakeFunction}
  FUNCTION Zero: INTEGER;
    BEGIN {Zero}
      WRITELN('enter Zero');
      Zero := 0;
      WRITELN('leave Zero')
    END; {Zero}
  BEGIN {Passing}
    TakeInteger(Zero);
    TakeFunction(Zero)
  END {Passing}.
```

Execution

```
OUTPUT:enter Zero
       leave Zero
       enter TakeInteger
                 0
       leave TakeInteger
       enter TakeFunction
       enter Zero
       leave Zero
                 0
       leave TakeFunction
```

Zero is entered and returns before TakeInteger is entered because Zero provides an integer value for IVar. However, TakeFunction is entered before Zero is called. It is not until WRITELN(IFunc) is executed within TakeFunction that Zero is entered. The WRITE statements within TakeInteger and TakeFunction have exactly the same form; to discover the meaning one must look at the declaration of the corresponding formal parameter.

23.2.3 Nonlocal References

Passing a function or procedure name for later invocation is a straightforward idea, but it raises a new possibility for confusion of variable names. In the following program, the variable V1 occurs both as a local that is in the execution state when a procedure parameter is invoked, and as a global that was in the state when that procedure was declared. One of these is altered, but which one?

```
PROGRAM NonLocal(OUTPUT);
  VAR
    V1: INTEGER;
    {Global variable available when ProcP declared}
  PROCEDURE ProcP(Val: INTEGER);
    BEGIN {ProcP}
       V1 := Val + 10
    END;  {ProcP}
  PROCEDURE ProcQ(PROCEDURE ProcParm(X: INTEGER));
    VAR
      V1: INTEGER;
      {Local variable available when ProcParm called}
    BEGIN {ProcQ}
      V1 := 2;
      ProcParm(V1);
      {If ProcParm is ProcP, which V1 is used?}
      WRITELN(V1)
    END; {ProcQ}
  BEGIN {NonLocal}
    V1 := 1;
    ProcQ(ProcP);
    WRITELN(V1)
  END. {NonLocal}
```

Execution

```
OUTPUT:          2
                12
```

The execution shows that the global `V1` is altered; that is, when a parameter procedure makes global references, they refer to the state at its declaration, not the state at its call. Thus the assignment

```
V1 := Val + 10
```

within `ProcP` changed the global `V1`, not the local of `ProcQ`, even though only the local was available before and after `ProcP` was called. This is another example of Pascal's *static* binding rule first encountered in Section 9.2.5. Identification of variables depends not on the dynamic declaration order at execution, but on the declaration order of the static program text.

23.2.4 Zero Finding using Bisection

`FUNCTION` parameters allow the creation of routines that operate not on single values, but on the whole collection that the function describes. The resulting software is a better encapsulation because it receives the function once as a parameter, and can use it as often as needed. If only functional values could be passed, it would be necessary to make repeated calls to obtain many values, and control would reside not in the routine using the function but in the program that passed it.

As an example, `Bisect` contains a routine `FindZero` that finds a zero of its function parameter. Passing a function parameter to `FindZero` means that `FindZero` can operate on any function, not just one whose definition is embedded inside; at the same time, the complete zero calculation is contained in the routine, and a single call to `FindZero` suffices. In the sample program, `FindZero` is supplied with `Plus1`, computing $f(x) = x+1$.

The idea used in `FindZero` is a variation of binary search. Suppose that the function supplied is monotonic—it steadily increases or decreases. If two argument values are supplied at which the function takes on values of different signs, then there must be a zero crossing between them. The interval is halved, and the zero confined to one half or the other, this process being repeated until the interval is so small that it effectively pins down the zero (as defined by a program constant `Epsilon`).

```
PROGRAM Bisect(OUTPUT);
  CONST
    Epsilon = 1E-5;
  VAR
    Z: REAL;
    Cnt: INTEGER;
  PROCEDURE FindZero(FUNCTION Func(X: REAL): REAL;
                Low, High: REAL;
                VAR Zero: REAL; VAR It: INTEGER);
    {Find a value for Zero such that
     Low<=Zero<=High and monotonic Func(X)=0
     for an X within Epsilon of Zero.  The
     number of bisections is returned in It.}
    LABEL
      4;
    BEGIN {FindZero}
      It := 0;
      Zero := Low; {in case loop body is not executed}
      If Func(Low) * Func(High) > 0.0
      THEN {no zero crossing for monotonic Func}
        BEGIN
          WRITELN('Bad endpoints supplied to FindZero');
          GOTO 4
        END;
      {Otherwise, proceed with binary search}
      WHILE ABS(High - Low) > Epsilon
      DO
        BEGIN
          It := It + 1;
          Zero := (High + Low) / 2;
          IF Func(Zero) < 0
          THEN
            IF Func(Low) < 0
            THEN
              Low := Zero
            ELSE
              High := Zero
          ELSE
            IF Func(High) >= 0.0
            THEN
              High := Zero
            ELSE
              Low := Zero
        END;
    4:
    END; {FindZero}
  FUNCTION Plus1(RArg: REAL): REAL;
    BEGIN {Plus1}
      Plus1 := RArg + 1.0
    END; {Plus1}
```

```
BEGIN {Bisect}
  FindZero(Plus1, -2.0, 2.0, Z, Cnt);
  WRITELN(Z:12:8, Epsilon:10:6, Cnt:4)
END. {Bisect}
```

Execution

```
OUTPUT:   -1.00000763   0.000010   19
```

The program required 19 iterations to find the zero to within 0.00001.

23.2.5 Sorting Different Kinds of Data

Many sorting algorithms have been investigated in this book, but they have been limited to character data as the simplest case. The following procedure makes use of FUNCTION and PROCEDURE parameters to allow the sorting of many different kinds of objects. The procedure uses an exchange sort, with the data stored in an array. The type of object sorted, the number of objects, and the methods for comparing and exchanging objects are all supplied external to the procedure Sort.

```
TYPE
  EltArray = ARRAY [1..NumElts] OF EltType;
PROCEDURE Sort(VAR Val: EltArray; NumToSort: INTEGER;
  FUNCTION Ordered(VAR A, B: EltType): BOOLEAN;
  PROCEDURE Swap(VAR X, Y: EltType));
  {Permute the elements Val[1], ..., Val[NumToSort]
   so that for all i,
   0<=i<NumToSort<=NumElts: Val[i]<=Val[i+1]}
  VAR
    Sorted: BOOLEAN;
    I, J: 1..NumElts;
  BEGIN {Sort}
    Sorted := False;
    I := NumToSort;
    WHILE (I >= 2) AND NOT Sorted
    DO
      BEGIN
        Sorted := TRUE;
        FOR J := 1 TO NumToSort-1 DO
          IF NOT Ordered(Val[J],Val[J+1])
          THEN
            BEGIN
              Swap(Val[J],Val[J+1]);
              Sorted := FALSE
            END;
        I := I - 1
      END
  END;  {Sort}
```

The user of Sort supplies the comparison and exchange routines for the parameters Ordered and Swap, a CONST declaration for NumElts, and a TYPE declaration for EltType.

To illustrate the use of Sort, two programs, SortStrings and SortDates, will be written to handle quite different data. To sort character strings, it is necessary to define a type to represent strings and operations to compare and exchange strings.

```
CONST
  NumElts = 5;
  EltLen = 10;
TYPE
  EltType = PACKED ARRAY [1..EltLen] OF CHAR;
FUNCTION Compare(VAR X, Y: EltType): BOOLEAN;
  BEGIN {Compare}
    Compare := X <= Y
  END;  {Compare}
PROCEDURE Exchange(VAR X, Y: EltType);
  VAR
    Temp: EltType;
  BEGIN {Exchange}
    Temp := X;
    X := Y;
    Y := Temp
  END;  {Exchange}
```

The driver program below reads and writes one string of characters per line, storing them in the array ToBeSorted.

```
PROGRAM SortStrings(INPUT, OUTPUT);
  {Include string type declarations and operations}
  VAR
    ToBeSorted: EltArray;
    TempElt: ARRAY [1..EltLen] OF EltType;
    I, J: 0 ..  NumElts;
  {Include PROCEDURE Sort(VAR Val: EltArray;
   NumToSort: INTEGER;
   FUNCTION Ordered(VAR A, B: EltType): BOOLEAN;
   PROCEDURE Swap (VAR X, Y: EltType))
    Permute the elements Val[1], ..., Val[NumToSort]
    so that for all i,
    0<=i<NumToSort<=NumElts: Val[i]<=Val[i+1]}
  BEGIN {SortStrings}
    WRITELN('The input:');
    {read and echo input}
    I := 0;
    WHILE (I < NumElts) AND NOT EOF
    DO
      BEGIN
        I := I + 1;
        J := 0;
        WHILE (J < EltLen) AND NOT EOLN
        DO
          BEGIN
            J := J + 1;
            READ(TempElt[J]);
            WRITE(TempElt[J])
          END;
        READLN;
        WRITELN;
        {blank fill the strings}
        FOR J := J+1 TO EltLen
        DO
```

```
            TempElt[J] := ' ';
          PACK(TempElt,1,ToBeSorted[I])
        END;
      Sort(ToBeSorted,I,Compare,Exchange);
      {write output}
      WRITELN('The sorted output:');
      FOR I := 1 TO I
      DO
        WRITELN(ToBeSorted[I])
  END.  {SortStrings}
```

Execution

```
INPUT :z01
       abc
       defg
       ab
OUTPUT:The input:
       z01
       abc
       defg
       ab
       The sorted output:
       ab
       abc
       defg
       z01
```

Date objects that represent dates in which months are abbreviated as in Section 14.1.2 can be sorted using Sort without many changes. The declarations are:

```
CONST
  NumElts = 15;
TYPE
  Month = (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);
  Date = RECORD
           Mo: Month;
           Day: 1..31
         END;
  EltType = Date;
```

for storing up to 15 Date values. The same Exchange function that moved strings will work for dates, but a new comparison routine is required:

```
FUNCTION Less(VAR D1, D2: Date): BOOLEAN;
  {Less := D1<D2 }
  BEGIN {Less}
    IF D1.Mo = D2.Mo
    THEN
      Less := D1.Day < D2.Day
    ELSE
      Less := D1.Mo < D2.Mo
  END;
```

These changes to convert from sorting strings to sorting dates reflect the care with which Sort was written. The test program was not designed to be easily reused, and it must be completely rewritten. The routines to read and write Month values come from Section 14.1.2.

```
PROGRAM SortDates(INPUT, OUTPUT);
  {Include date type declarations}
  VAR
    ToBeSorted: EltArray;
    I: 0..MAXINT;
  {Include ReadMonth, WriteMonth, Less, Exchange, Sort}
  BEGIN {SortDates}
    WRITELN('The input:');
    {read and echo input}
    I := 0;
    WHILE (I < NumElts) AND NOT EOF
    DO
      BEGIN
        I := I + 1;
        ReadMonth(INPUT,ToBeSorted[I].Mo);
        READ(ToBeSorted[I].Day);
        READLN;
        WriteMonth(OUTPUT,ToBeSorted[I].Mo);
        WRITE(ToBeSorted[I].Day:3);
        WRITELN
      END;
    Sort(ToBeSorted,I,Less,Exchange);
    {write output}
    WRITELN('The sorted output:');
    FOR I := 1 TO I
    DO
      BEGIN
        WriteMonth(OUTPUT,ToBeSorted[I].Mo);
        WRITE(ToBeSorted[I].Day:3);
        WRITELN
      END
  END.  {SortDates}
```

Execution

```
INPUT :FEB 1
       JAN 29
       JAN 30
       JAN 25
OUTPUT:The input:
       FEBRUARY  1
       JANUARY 29
       JANUARY 30
       DECEMBER 25
       The sorted output:
       JAN 29
       JAN 30
       FEB  1
       DEC 25
```

23.2.6 Exercises

23.2.1 Given the syntactically correct procedure statement

```
CallItAndPassIt(Fun(X))
```

but no access to the declarations of the identifiers, give as much information as is certain about the parameter of `CallItAndPassIt`.

23.2.2 Use box functions to give a formal meaning for PROCEDURE parameters, as was done (for example) for value parameters in Section 15.4. Confine the definition to the simplest case of a procedure with one procedure parameter, which itself has no parameters.

23.2.3 Give the output produced by the following program, and explain how it comes about.

```
PROGRAM P(OUTPUT);
  VAR
    Z: INTEGER;
  PROCEDURE T(PROCEDURE U);
    BEGIN
      Z := Z - 2;
      U
    END;
  PROCEDURE Q(PROCEDURE R(PROCEDURE S);
                VAR X: INTEGER);
    VAR
      Y: INTEGER;
    PROCEDURE V;
      BEGIN
        X := X + 1
      END;
    BEGIN
      Y := 2;
      X := Z - Y;
      R(V)
    END;
  BEGIN
    Z := 6;
    Q(T,Z);
    WRITELN(Z)
  END.
```

23.2.4 Give the output produced by the following program, and explain how it comes about.

```
PROGRAM P(OUTPUT);
  VAR
    A: INTEGER;
  PROCEDURE A(PROCEDURE T(VAR X: INTEGER)); FORWARD;
  PROCEDURE Q(VAR B: INTEGER);
    VAR
      C: INTEGER;
    BEGIN
      C := B+A;
      WRITELN(C);
      IF B > 0
      THEN
        S(Q)
    END;
  PROCEDURE S;
    VAR
      C: INTEGER;
    BEGIN
      C := 0;
      T(C)
    END;
  BEGIN
```

```
    A := 7;
    Q(A)
END.
```

23.2.5 Write a procedure with a function parameter that writes a table of arguments and function values, where the arguments to print are supplied by three parameters giving starting and stopping values, and the number of points between.

23.2.6 Write a procedure with a function parameter that produces a graph of the parameter function. The X and Y ranges are also to be supplied as parameters.

23.2.7 Use `FindZero` (Section 23.2.4) to write a program that calculates square roots. The input is a `REAL` value z, and the square root is obtained by finding a zero for the function $f(x) = x^2 - z$.

23.2.8 Eliminate the GOTO statement from `FindZero` (Section 23.2.4).

23.2.9 Rewrite `FindZero` (Section 23.2.4) as a recursive procedure.

23.2.10 Can the value of `Epsilon` in `FindZero` (Section 23.2.4) be made so small that a zero will never be found? Hint: suppose that there is a good deal of arithmetic subject to roundoff error in computing `Func`.

23.2.11 Rewrite `FindZero` (Section 23.2.4) as a procedure that need not be given two points between which to search for the zero of a monotonic function.

a) Give it only one point at which to begin.

b) Give it no information about where to look.

Hint: A kind of inverse of binary search moves outward from a point, doubling the interval on each side. How important is the restriction to monotonic functions?

23.2.12 Describe some of the possibilities that can occur if `FindZero` (Section 23.2.4) is given an actual parameter that is a function which is not monotonic.

23.2.13 Describe the changes needed for the `Sort` procedure of Section 23.2.5 (not the driver program) to be able to sort

a) `REAL` values

b) `CHAR` values

c) `SET OF CHAR` values

d) `TEXT` file values

23.2.14 Are there data types that can be sorted by `Sort` (Section 23.2.5) for which the `Exchange` procedure written for the string type would not be adequate? Explain.

23.2.15 Design a driver program for `Sort` that is more amenable to change than those given in Section 23.2.5, and show how it could handle the two examples of that section.

23.2.16 Design a `MergeSort` program (Section 19.3.2) that is as easy to change as possible.

23.3 Variant Records

Preview

Information cannot always be divided into one set of fixed fields to form a Pascal record. When different formats could be conveniently mixed, the variant record provides a type to encompass them all.

A record binds together several different types as its fields. The record types presented beginning in Chapter 16 have each had a fixed list of types; that is, each value had the same fields. Real information is often not so simple. For example, in a collection of personnel records (name, address, social security number, etc.) there will be some fields that are appropriate for only some employees. Not all people will be on yearly salaries; instead some records will need fields for hourly

pay rate and the number of hours worked. Many operations on these data (e.g., correcting the spelling of a name, or producing a sorted list of employees) do not depend on the information in such "variant" fields. Instead of enlarging the record to contain all possible different fields, or separating employees into groups with similar fields, *variant records* can be used to make one TYPE for records with varying content. An employee variant record might be described as follows:

```
TYPE
  Month = (NoMonth, Jan, Feb, Mar, Apr, May, Jun,
           Jul, Aug, Sep, Oct, Nov, Dec);
  Date = RECORD
           Mo: Month;
           Day: 1..31;
           Year: INTEGER
         END;
  Compensation = (Salary, Hourly);
  Person = RECORD
             Name: PACKED ARRAY[1..Len] OF CHAR;
             SocialSecurity: INTEGER;
             Address: PACKED ARRAY[1..Len] OF CHAR;
             BirthDate: Date;
             CASE WageType: Compensation OF
               Hourly: (Hours: 0 .. 168;
                        HourlyRate: REAL);
               Salary: (YearlyRate: 0 ..  MAXINT)
           END;
```

The variant part of the record is the CASE construction at the end (and to make sure it appears last, it is terminated by the same END that terminates the RECORD.) The fields that are not part of every record are presented as alternatives in the CASE, and a finite ordinal type (here Compensation) provides a list of identifiers (Hourly, Salary) to mark the variant parts. This record contains an implicit field WageType of type Compensation called the *tag*, and the record is said to be *discriminated* by this tag.

Person is like a *union* of two different record types:

```
TYPE
  HPerson = RECORD
              Name: PACKED ARRAY[1..Len] OF CHAR;
              SocialSecurity: INTEGER;
              Address: PACKED ARRAY[1..Len] OF CHAR;
              BirthDate: Date;
              WageType: Compensation
              Hours: 0 .. 168;
              HourlyRate: REAL
            END;
  SPerson = RECORD
              Name: PACKED ARRAY[1..Len] OF CHAR;
              SocialSecurity: INTEGER;
              Address: PACKED ARRAY[1..Len] OF CHAR;
              BirthDate: Date;
              WageType: Compensation;
              YearlyRate: 0 ..  MAXINT
            END;
```

A variable of type Person is allocated enough storage to hold values of the larger of the two types HPerson and SPerson so that Hours and HourlyRate occupy the same relative position as YearlyRate.

The advantage of having a single type with variants instead of two different types is that a single routine can manipulate type Person instead of one routine for an HPerson and another for an SPerson. Within such a routine, the tag field WageType is used to choose which portion of the record is valid, as in the following example:

```
VAR
  P: Person;
BEGIN
  ...
  CASE P.WageType OF
    Hourly: WRITELN(P.Hours * P.HourlyRate);
    Salary: WRITELN(P.YearlyRate DIV 52)
  END;
  ...
END
```

The CASE statement selects statements in which the fields match the similar part of the CASE construction in the record declaration. Care must be taken to set the correct tag value in any value of type Person. The tag is just like other fields; it has no initial value and does not automatically adjust when the variant fields are changed. Strange results can be obtained by inconsistently manipulating the tag and the variant fields, as in:

```
P.WageType := Hourly;
P.YearlyRate := 35000;  {P.WageType is still Hourly}
CASE P.WageType OF      {execute the Hourly case
                         with unpredictable results}
  Hourly: WRITELN(P.Hours * P.HourlyRate);
  Salary: WRITELN(P.YearlyRate DIV 52)
END;
```

The variant part of a record need not have a tag at all—then it is said to be *free*. The fields of a free variant are expected to occupy the same storage space in a particular Pascal machine, and provide a way to circumvent the type restrictions of the language. For example, if a programmer knows that a CHAR variable and an INTEGER variable occupy the same space, then the INTEGER value of A can be printed as follows:

```
PROGRAM Convert(OUTPUT);
  TYPE
    CvtType = (Int, Chr);
    Cvt = RECORD
            CASE CvtType OF
              Int: (IVal: INTEGER);
              Chr: (CVal: CHAR)
          END;
  VAR
    Both: Cvt;
  BEGIN
    Both.CVal := 'A';
    WRITELN(Both.IVal)
  END.
```

Execution

```
OUTPUT:          65
```

Variant records with free variants are a feature of Pascal that should be avoided, since they cannot be expected to work in the same way on different Pascal machines.

23.3.1 Variant Record Syntax

The syntax rules for variant records are shown below.

<unpacked structured type> ::= RECORD <field list> END

<field list> ::= <fixed part> <variant part> | <fixed part>
| <variant part> |

<fixed part> ::= <record sections>

<record sections> ::= <record sections> ; <record section> | <record section>

<record section> ::= <identifier list> : <type denoter>

<variant part> ::= CASE <variant selector> OF <variant list>

<variant selector> ::= <optional tag field> <ordinal type identifier>

<optional tag field> ::= <identifier> : |

<variant list> ::= <variant list> ; <variant> | <variant>

<variant> ::= <constant list> : (<field list>)

23.3.2 Pointers to Variant Records

Pointers can reference an object whose type is a variant record. For example:

```
TYPE
  Compensation = (Salary, Hourly);
  PList = ↑Person;
  Person = RECORD
             Name: PACKED ARRAY[1..Len] OF CHAR;
             SocialSecurity: INTEGER;
             Address: PACKED ARRAY[1..Len] OF CHAR;
             BirthDate: Date;
             Next: PList;
             CASE WageType: Compensation OF
               Hourly: (Hours: 0 .. 168;
                        HourlyRate: REAL);
               Salary: (YearlyRate: 0 ..  MAXINT)
           END;
VAR
  First: PList;
```

Storage for a `Person` pointed to by `First` can be obtained with `NEW(First)` as usual. The resulting `Person` target will be able to have either an `Hourly` or `Salary` variant part. However, if the allocation is made with `NEW(First,Salary)`, it is expected that the variant will never be `Hourly`. In this case, since the `Hourly` variant is probably the larger, strange things may happen if the programmer is not careful to set the tag to `Salary`. When there are more than two tag values, any sequence of them may be arguments to `NEW`, but they must occur in the order of the declaring `CASE`. Only enough storage for the largest type listed is guaranteed. If tag constants are used in a NEW statement to obtain storage, the same constants must be used in a DISPOSE statement to return the storage.

23.3.3 Concordance

A concordance for a text is a list of the words that appear in the text, and count of how often each word appears. Common words like articles and pronouns are usually omitted from the concordance (they are said to be *blocked*). Thus a concordance program could be written using a list of records with a string field for the word and an integer field for the count. A frequent operation in computing a concordance is looking up a word to see if it is blocked. Thus searching the blocked list for a word should be a fast operation, and since the list can be large, the search time should depend as little as possible on its size. A *hash table* is a storage structure with these properties.

A *hash function* is a transformation mapping data values to hash table indices. A good hash function spreads the data items over the hash table uniformly so that looking up an item is no more expensive than an array reference. However, if two data items map to the same hash table index, a *collision* occurs, and something must be done to resolve the conflict. The simplest collision-resolving algorithm is to keep all the data items with the same hash-function value in a list called a *bucket*.

`Concordance` will use a hash table `HashT` and a list of words `WordList` whose types will be defined as part of later refinements. The text to be processed is in the input file; `Common` is an external file containing the blocked words.

Design Part 1

```
PROGRAM Concordance(INPUT,OUTPUT,Common);
  {Declare:
    Common - blocked-word file
    HashT - internal table of blocked words
    WordList - concordance of nonblocked words}
  BEGIN {Concordance}
    HashInit(HashT);
    {read words from Common and insert in HashT};
    ListInit(WordList);
    WHILE NOT EOF
    DO
      BEGIN
        {read a word from INPUT;  a word not in Common
         or WordList should be added to WordList with
         1 occurrence, while a word not in Common but
         already in WordList should have an extra
         occurrence recorded}
      END;
    {print the words and their occurrences}
    ListPrint(WordList)
  END.  {Concordance}
```

Procedure `ReadWord` from Section 19.4.1 can be used to move the next word from `Common` to the local variable `Temp`, and to set `Found` to TRUE if a word was read. `HashInsert` installs the word just read in `HashT`.

Design Part 1.1

```
{read words from Common and insert in HashT}
RESET(Common);
WHILE NOT EOF(Common)
DO
  BEGIN
    ReadWord(Common,Temp,Found);
    IF Found
    THEN
      HashInsert(HashT,Temp)
  END;
```

ReadWord can also be used to obtain words from INPUT. HashProbe is a function that returns TRUE if a word occurs in a hash table, and FALSE if it does not.

Design Part 1.2

```
{read a word from INPUT;  a word not in Common
 or WordList should be added to WordList with
 1 occurrence, while a word not in Common but
 already in WordList should have an extra
 occurrence recorded}
WHILE NOT EOF
DO
  BEGIN
    ReadWord(INPUT,Temp,Found);
    IF Found AND NOT HashProbe(HashT,Temp)
    THEN
      ListInsert(WordList,Temp)
  END
```

The declarations for HashT and WordList can be given now that the processing algorithm has been fixed. WordList is a list of nodes, each containing a word and an integer. HashT is essentially an array of word lists, so the same procedures can be used to maintain it that maintain WordList, by making them variants of the same RECORD type. Although each element of a word list needs an integer (Occurs) to count how often the word appears, in a hash table element this component is missing.

Design Part 1.3

```
{Declare:
  Common - blocked-word file
  HashT - internal table of blocked words
  WordList - concordance of nonblocked words}
CONST
  Max = 20; {Longest word}
  TSize = 17; {Size of the hash table}
TYPE
  WordType = PACKED ARRAY[1..Max] OF CHAR;
  NodeType = (HTable, WList);
  List = ↑Node;
  Node = RECORD
           Next: List;
           Word: WordType;
           CASE Tag: NodeType OF
             HTable: ();
             WList: (Occurs: 0..MAXINT)
         END;
```

```
  HTableType = ARRAY[0..TSize] OF List;
VAR
  Common: TEXT;
  HashT: HTableType;
  WordList: List;
  Temp: WordType;
  Found: BOOLEAN;
```

A single routine to process HashT elements and WordList reduces the size of the concordance program, but its more important advantages are in debugging and maintenance. Improvements or corrections in the list insertion algorithm benefit both structures, and any changes can be made in just one routine.

ListInit assigns the value NIL to the pointer that references the first item in a list.

Design Part 1.4

```
PROCEDURE ListInit(VAR L: List);
  { L := <>}
  BEGIN {ListInit}
    L := NIL
  END;  {ListInit}
```

HashInit simply calls ListInit once for each of its table entries.

Design Part 1.5

```
PROCEDURE HashInit(VAR T: HTableType);
  { T := [] }
  VAR
    I: 0..TSize;
  BEGIN {HashInit}
    FOR I := 1 TO TSize
    DO
      ListInit(T[I])
  END;  {HashInit}
```

ListInsert obtains an address for the element that is being inserted, sets the element's Tag field to indicate the proper variant, and (based on whether or not the element was already in the list) initializes or updates the element's Occurs field.

Design Part 1.2.1

```
PROCEDURE ListInsert(VAR L: List; VAR Txt: WordType);
{If Txt is in L, then increase the number of its
 occurrences by 1.  Otherwise add a new member to L
 whose Word field has the same value as Txt and whose
 Occurs field shows a single occurrence.}
 VAR
   WdPtr: List;
   Found: BOOLEAN;
 BEGIN {ListInsert}
   Insert(L,WdPtr,Txt,Found);
   WdPtr↑.Tag := WList;
   IF NOT Found
   THEN
     WdPtr↑.Occurs := 1
   ELSE
     WdPtr↑.Occurs := WdPtr↑.Occurs + 1
 END;  {ListInsert}
```

The design of the insertion routine is a challenge—how much common processing can be done for word lists and hash buckets? Both are free of duplicate entries, but a word-list element has the extra `Occurs` component to process. If the word to be be inserted is already on the list, `Insert` returns a pointer to it; otherwise, a new list member is created and a pointer to it returned. `Found` can be used to distinguish these cases.

Design Part 1.2.1

```
PROCEDURE Insert(VAR L, Elt: List;
                 VAR Txt: WordType;
                 VAR Found:BOOLEAN);
{L is a list of n records, one of whose fields,
 Word, is a WordType value.
   (If Txt has the same value as the Word field of
    the node at position j  -->
        Elt,Found := j,TRUE) |
   (If Txt does not occur as any Word field in L -->
        L,Elt,Found :=
           L with Txt appended in position n+1,
            n+1, FALSE) }
  VAR
    Prev, Curr: List;
  BEGIN {Insert}
    {(Txt matches the Word field of the element
      at position j in L -->
      Found,Prev,Curr :=
          TRUE, address of L's element whose Next
           field contains the address of L's j'th
           element, address of L's j'th element) |
     (Txt matches none of the Word fields of L -->
      Found,Prev,Curr :=
          FALSE, address of L's last element, NIL )};
    IF Found
    THEN
      Elt := Curr
    ELSE
      BEGIN
        NEW(Elt);
        Elt↑.Word := Txt;
        {insert Elt after Prev↑}
      END
  END;  {Insert}
```

As in Section 20.2.3, two pointer variables `Prev` and `Curr` are used to search for a word. If the search is unsuccessful, `Prev` is made to point to the last element in the list, where an insertion may be made.

Design Part 1.2.1.1

```
{(Txt matches the Word field of the element
  at position j in L -->
  Found,Prev,Curr :=
       TRUE, address of L's element whose Next
        field contains the address of L's j'th
        element, address of L's j'th element ) |
 (Txt matches none of the Word fields of L -->
  Found,Prev,Curr :=
       FALSE, address of L's last element, NIL )};
Found := FALSE;
Prev := NIL;
Curr := L;
WHILE (Curr <> NIL) AND NOT Found
DO
  IF Curr↑.Word = Txt
  THEN
    Found := TRUE
  ELSE
    BEGIN
      Prev := Curr;
      Curr := Curr↑.Next
    END;
```

When a new element is to be inserted, either the list is empty and it becomes the first element, or `Prev` points to where it should be inserted.

Design Part 1.2.1.2

```
{insert Elt after Prev↑}
IF L = NIL
THEN {insert as first element}
  L := Elt
ELSE
  BEGIN {insert as element after Prev↑}
    Elt↑.Next := Prev↑.Next;
    Prev↑.Next := Elt
  END
```

`HashInsert` uses the function `Hash` to determine the bucket in which the new word should be inserted. Then it simply sets the `Tag` field of the new element to the appropriate value.

Design Part 1.1.1

```
PROCEDURE HashInsert(VAR T: HTableType;
                     VAR Txt: WordType);
  { T := T union [Txt] }
  VAR
    WdPtr: List;
    Occurs: BOOLEAN;
  BEGIN {HashInsert}
    Insert(T[Hash(Txt)],WdPtr,Txt,Occurs);
    WdPtr↑.Tag := HTable
  END;  {HashInsert}
```

A simple hash function adds the integer representation of the first two characters of a word and takes the modulus of this sum with the size of the table. This is a particularly poor hash function if many blocked words start with the same letters (e.g., the, this, these, those, they, etc.) because such words wind up in the same (long) bucket.

Design Part 1.1.1.1

```
FUNCTION Hash(VAR Txt:WordType): INTEGER;
  {Hash := an integer in the range 0..TSize-1}
  BEGIN {Hash}
    Hash := (ORD(Txt[1])+ORD(Txt[2])) MOD TSize;
  END;  {Hash}
```

HashProbe searches the hash table to determine if a word is there, using Hash to find the proper bucket.

Design Part 1.6

```
FUNCTION HashProbe(VAR T: HTableType;
                   VAR Txt: WordType): BOOLEAN;
  { HashProbe := Txt is a member of T }
  VAR
    P: List;
    Index: 0..TSize;
  BEGIN {HashProbe}
    HashProbe := FALSE;
    Index := Hash(Txt);
    IF T[Index] <> NIL
    THEN
      BEGIN
        P := T[Index];
        WHILE (P <> NIL)
        DO
          BEGIN
            IF P↑.Word = Txt
            THEN
              HashProbe := TRUE;
            P := P↑.Next
          END
      END
  END;  {HashProbe}
```

Finally, ListPrint prints the word list as the output from Concordance. Since it might be necessary to print the hash table during development, this procedure checks each element's Tag field before trying to print the number of occurrences.

Design Part 1.7

```
PROCEDURE ListPrint(L: List);
  {write the contents of L to OUTPUT}
  VAR
    Ptr: List;
  BEGIN {ListPrint}
    WRITELN('List values are:');
    Ptr := L;
    WHILE Ptr <> NIL
    DO
      BEGIN
        WRITE(Ptr↑.Word);
        IF Ptr↑.Tag = WList
        THEN
          WRITE(Ptr↑.Occurs);
        WRITELN;
        Ptr := Ptr↑.Next
```

```
    END
  END;  {ListPrint}
```

The development of Concordance is complete and the parts can be assembled. If we decided that the concordance should be printed in sorted order, Insert could be changed to maintain a word list in sorted order. As a beneficial side effect of such a change, the buckets of the hash table would also be sorted and the search in HashProbe could be rewritten to improve its performance.

23.3.4 Exercises

23.3.1 In the Person example of Section 23.3, why would two different procedures need to be written to process the types HPerson and SPerson?

23.3.2 A procedure Draw is to be written to handle objects representing points, lines, and circles. Assume that a point is represented by a pair of coordinates (two INTEGER values), a line by two points, and a circle by a point and an INTEGER radius.

a) Give the type declarations needed to permit Draw to use a single parameter whose value might be any of these types.

b) Give a skeleton of the code within Draw that would use the parameter to choose which part of the code to execute. Suggest some parts of the processing in Draw that would benefit from having just one routine for all three objects.

23.3.3 A programmer claims he can print the INTEGER representation of the BOOLEAN value TRUE with the following program fragment:

```
new(Z);
Z↑.Bool := TRUE;
WRITELN(Z↑.Int)
```

If this can be done, give the type definitions that make this fragment work. Otherwise, explain why this is not possible.

23.3.4 Write a procedure GetToken that scans a TEXT file and returns a series of tokens (words, integers, punctuation marks, and an end token). A token is represented as a variant record; the punctuation and end token have empty variant fields.

```
PROGRAM Scan(INPUT,OUTPUT);
{break input into a series of tokens}
  CONST
    MaxWordLength = 20;
  TYPE
    Word = PACKED ARRAY [1..MaxWordLength] OF CHAR;
    TokenType = (Txt, Int, EoT, Period, QMark,
                 XMark, Comma, Colon, Semicolon,
                 Undefined);
    Token = RECORD
              CASE T: TokenType OF
                Txt: (CValue: Word);
                Int: (IValue: INTEGER);
                EoT, Period, Comma, QMark, XMark,
                Colon, Semicolon, Undefined: ()
            END;
  PROCEDURE GetToken(VAR Symbol: Token);
    {Fill in the code};
```

```
BEGIN {Scan}
  GetToken(Tok);
  WHILE Tok.T <> EoT DO
    BEGIN
      CASE Tok.T OF
        Undefined: WRITELN('Undefined token');
        Txt: WRITELN('Word = ''',Tok.CValue,'''');
        Int: WRITELN('Number =',Tok.IValue:5);
        Period: WRITELN('Period');
        Comma: WRITELN('Comma');
        Colon: WRITELN('Colon');
        Semicolon: WRITELN('Semicolon');
        XMark: WRITELN('Exclamation mark');
        QMark: WRITELN('Question mark')
      END;
      GetToken(Tok)
    END
END.  {Scan}
```

Your program should produce output like that shown below.

Execution

```
INPUT :Number 54.67! ,asdfjkl#;:
OUTPUT:Word    'Number
       Number      54
       Period
       Number      67
       Exclamation mark
       Comma
       Word    'asdfjkl
       Undefined token
       Semicolon
       Colon
```

23.3.5 Modify the `Concordance` program of Section 23.3.3 so that at the end of execution it prints statistics about the quality of the hashing, notably the number of collisions, and the size and contents of the hash buckets.

23.3.6 Modify the `Insert` procedure of `Concordance` (Section 23.3.3) so that it keeps sorted word lists. Also modify `HashProbe` to improve its speed.

23.4 Chapter Summary

Judicious use of `CONST` and `TYPE` declarations and `PROCEDURE` parameters makes Pascal programs more flexible. Altering a single declaration or the value of an actual parameter can then change the meaning substantially, but in a controlled way. Variant records factor data into disjoint parts and cluster the similar parts at the head of a composite record. An operation that uses only the common fields can then be used to centralize all record processing. The benefits of single-procedure implementation are enormous when it becomes necessary to extend the operation or repair an error—only one copy of the operation needs to be changed.

CHAPTER 24

ONE LAST SOLUTION

Chapter Preview

The solution to a final problem illustrates the design and analysis methods of this text. By using analysis within the design, confidence in the program is maintained as it grows. The design method scales up to problems of arbitrary size by decomposing large problems into a hierarchy of smaller ones, and maintaining intellectual control of the components. Complete formality cannot be sustained for a large problem, but formal methods can be selectively applied to problems of any size.

This book has introduced two major design strategies that complement and build on one another. The first is stepwise refinement of Pascal program parts, particularly Pascal statements; at each step a relational or functional specification is refined into a small design and further subspecifications. The second is designing modules with data abstractions to gain intellectual control. When carefully chosen, such modules can be reused as building blocks for other designs.

This chapter contains an integrated example that illustrates both stepwise refinement and data abstraction. The example, to find and print a set of prime numbers, is large enough to require the judicious use of formal proofs, embedded in a broader activity of informal reasoning and design. In particular, the example illustrates that stepwise refinement permits the control of details, step by step, in both data and operations. For example, a subspecification need only treat variables that are used in the design it supports and not the variables that are introduced for its implementation. Deferring details while maintaining control of the design is essential in scaling up methods of "programming in the small" to "programming in the large." Both stepwise refinement and data abstraction are vital in scaling up. Stepwise refinement permits the deferring of details directly as a program is elaborated a part at a time. Data abstraction permits the deferring of details indirectly, but with even more power, by increasing the capability of the underlying design primitives.

24.1 The Prime Number Problem Again

Preview

The problem: to find and print the prime numbers up to a given integer.

The set of prime numbers up to natural number N is formally defined as:

$$\{P: 2 \leq P \leq N, \text{ P is a prime}\}.$$

The condition "P is a prime" means, of course, that P is not divisible by any natural number except 1 (which divides any natural number). That condition can be elaborated in a redefinition of the set of primes:

$$\{P : 2 \leq P \leq N, \text{ for no } Q > 1 \text{ is P mod Q} = 0\}.$$

This problem was solved in Chapter 16 in a direct way, using the Pascal SET type.

More information about primes and nonprimes is needed for the solution of this chapter. One useful property of primes is the fact that if Q divides P, than any divisor of Q also divides P. Thus small primes can be used to check P as a potential prime and their multiples need not be tried. There are many multiples of small primes (compared to N) so the savings can be significant.

The value of using small primes to check if large numbers are prime leads to a design strategy of finding the primes in order from small to large. Each prime found can then be used to save work in checking larger numbers. For this reason, the primes will be represented as a list of numbers

rather than a set. As each new prime is found, it will be added to the list.

24.1.1 Developing and Proving the Solution

Let `Primes` be the list of prime numbers up to `INTEGER N`. To calculate `Primes`, the following observation is helpful: 2 is the first prime and the only even prime. All other primes are odd, so only odd numbers need be tested as additional prime candidates. Under these conditions the problem statement can be refined to:

Design Part 1

```
{Compute Primes}
{Primes := <2>};
{Primes := Primes & OddPrimes(3,N)};
{print Primes}
```

`OddPrimes` is defined as follows:

$$\texttt{OddPrimes(L,U)} = \begin{cases} \texttt{<L: Prime?(L)> \& OddPrimes(L+2,U) if L} \leq \texttt{U and Odd(L)} \\ \texttt{<> if L>U} \end{cases}$$

$$\texttt{Prime?(X)} = \begin{cases} \texttt{TRUE if } \forall \texttt{P, P is in OddPrimes(3,X-1), X mod P} \neq 0 \\ \texttt{FALSE otherwise} \end{cases}$$

This definition allows the refinement:

Design Part 1.1

```
{Primes := Primes & OddPrimes(3,N)}
NextP := 3;
{Odd(NextP) → Primes := Primes & OddPrimes(NextP, N)}
```

Design Part 1.1.1

```
{Odd(NextP) → Primes := Primes & OddPrimes(NextP, N)}
WHILE NextP <= N
DO
  {(Primes,NextP := Primes & <NextP: Prime?(NextP)>,NextP+2)}
```

If the design is to remain under intellectual control at this point, it is necessary to prove that the WHILE statement in Design Part 1.1.1 is correct. That is, if

f = `(Odd(NextP) → Primes := Primes & OddPrimes(NextP, N))`

and W is:

```
WHILE NextP <= N
DO
  {(Primes,NextP := Primes & <NextP: Prime?(NextP)>,NextP+2)}
```

then the three conditions of the WHILE statement verification rule are met.

f and $\boxed{W}$ cannot be identical since `NextP` is altered in the loop body of W but is not changed in the concurrent assignment for f. `NextP` is not a part of f because its value is not of interest after the iteration is complete. Omitting `NextP` from the verification process eliminates details introduced by purely local variables. With this understanding we turn to the three steps of the proof.

1. domain(f) = domain($\boxed{W}$)

f is defined for all odd numbers. For these numbers W terminates immediately if `NextP>N`. If `NextP<=N`, the body of W is executed and `NextP` incremented assuring the eventual termination of the statement. We are not interested in the even numbers for which f is not defined, because the initialization assignment to `NextP` and advancing by 2 each time eliminates them from the iteration.

2. (NextP > N → f) = (NextP > N →)

The definition of OddPrimes assures us that if NextP>N then

OddPrimes(NextP, N) = <>.

Thus *f* reduces to

(NextP>N and Odd(NextP) → Primes := Primes & <>)

which is an identity function with a smaller domain than the identity function on the right side of step 2. Again, the even case is not of interest.

3. *f* =

((NextP ≤ N →
Primes,NextP := Primes & <NextP: Prime?(NextP)>,NextP+2) |
(NextP > N →)) ∘ *f*

The composition can be studied with a trace table:

Condition	Primes	NextP
NextP≤N	Primes & <NextP : Prime?(NextP)>	NextP+2
Odd(NextP+2)	Primes & <NextP : Prime?(NextP)> & OddPrimes(NextP+2,N)	

This trace table computes the part function:

Odd(NextP+2) →
Primes := Primes & <NextP: Prime?(NextP)> & OddPrimes(NextP+2,N)

NextP+2 is odd if NextP is odd and even if NextP is even, so:

Odd(NextP+2) ≡ Odd(NextP)

When NextP is odd:

(<NextP: Prime?(NextP)> & OddPrimes(NextP+2,N)) = OddPrimes(NextP,N)

(NextP+1 cannot be prime because it is an even integer.) Thus the simplified part function is:

NextP≤N and Odd(NextP) → Primes := Primes & OddPrimes(NextP,N)

The other case resulting from this composition is:

NextP>N and Odd(NextP) → Primes := Primes & OddPrimes(NextP,N)

Combining these two part functions yields:

(NextP≤N and Odd(NextP) → Primes := Primes & OddPrimes(NextP,N)) |
(NextP>N and Odd(NextP) → Primes := Primes & OddPrimes(NextP,N))

which is identical to *f*.

The body of the WHILE statement is refined next.

Design Part 1.1.1.1

```
{(Primes,NextP := Primes & <NextP: Prime?(NextP)>,NextP+2)}
{IsPrime := (For all members P of Primes, NextP mod P <> 0)};
IF IsPrime
THEN
  Append(Primes,NextP);
NextP := NextP + 2
```

Append will be defined as an operation on the abstract data type "list of integers" to add the value of NextP to the end of the list Primes. This is equivalent to concatenating to Primes the single element list containing NextP.

```
PROCEDURE Append(VAR L: List; Elt: EltType);
  {abs: (Length(L) < MaxSize --> L := L & <Elt>) |
        (Length(L) >= MaxSize --> )
```

The initial value of IsPrime is TRUE, which is the value of

```
(For all members P of Primes, NextP mod P <> 0)
```

if Primes is empty. The elements of Primes are examined sequentially to determine if any of them divide NextP, and the results of these tests are combined with previous test results in IsPrime.

Design Part 1.1.1.1.1

```
{IsPrime := (For all members P of Primes, NextP mod P <> 0)}
IsPrime := TRUE;
Head(Primes, Trial);
{(Primes = (<P1,...,Pi>, <Pi+1,...,Pn>) and Trial = Pi and i>=1 -->
     IsPrime := IsPrime and (NextP mod Trial <> 0) and
                (for all Pj in <Pi+1,...,Pn>, NextP mod Pj <> 0)) |
 (Trial = EndList --> )}
```

Operations must be added to the abstract data type so that values of list elements can be obtained sequentially, just as characters are read from a TEXT file. Like RESET for a file, Head(Primes,Trial) assigns Trial the value of the first element in Primes (if Primes is not empty) and the value EndList otherwise. In addition, Primes is prepared for reading, which we indicate by writing its elements as

```
<already-read values of Primes> & <to-be-read values of Primes>.
```

The other operation, Next(Primes,Trial), performs a function like that of the READ statement, assigning the next value to be read in Primes to Trial and advancing the reading pointer. If no more values remain to be read from Primes, Next assigns Trial the value EndList. Formal specifications allow reasoning about these operations.

```
PROCEDURE Head(VAR L: List; VAR Result: INTEGER);
  {abs: (L = <> --> Result := EndList) |
        (L = <L1,...,Li-1> & <Li,...,Ln> -->
            L, Result := <L1> & <L2,...,Ln>, L1)}
PROCEDURE Next(VAR L: List; VAR Result: INTEGER);
  {abs: (L = <L1,...,Ln> & <> --> Result := EndList) |
        (L = <L1,...,Li-1> & <Li,...,Ln> -->
            L, Result := <L1,...,Li> & <Li+1,...,Ln>, Li)}
```

Returning to the design part, the domain guard

```
Primes = <P1,...,Pi> & <Pi+1,...,Pn> and Trial = Pi and i>=1
```

implies that at least one element of Primes has already been read, that its value is the same as that of Trial, and that its value differs from EndList (which must be a value that differs from all list values). We show that the design part computes its specified function. If Primes is empty, Head assigns Trial the value EndList. The first condition in the following comment must be false because Primes contains no element with value EndList. Since Trial=EndList, the comment specifies an identity function that leaves IsPrime with the value TRUE in the first assignment statement. The value of IsPrime is thus identical to

```
(For all members P of Primes, NextP mod P <> 0)
```

which is vacuously true when Primes is empty. If Primes is not empty, Head assigns Trial a value different from EndList, so we need only consider the first of the two cases in the comment.

Condition	IsPrime	Trial	Primes
	TRUE		
Primes = <P1,...,Pn>		P1	<P1> & <P2,...,Pn>
<P1> & <P2,...,Pn> = <P1,...,Pi> & <Pi+1,...,Pn> and P1 = Pi and i>=0	TRUE and (NextP mod P1 <> 0) and (for all Pj in <Pi+1,...,Pn>, NextP mod Pj <> 0)		

Condition:

```
Primes = <P1,...,Pn> AND
<P1> & <P2,...,Pn> = <P1,...,Pi> & <Pi+1,...,Pn> AND
P1 = Pi and i >= 1
```

Picking `i=1` simplifies the condition to `Primes = <P1,...,Pn>`.

Assignment to `IsPrime`:

```
TRUE AND (NextP mod P1 <> 0) AND
  (for all Pj in <Pi+1,...,Pn>, NextP mod Pj <> 0)
```

which is

```
TRUE AND (NextP mod P1 <> 0) AND
  (for all Pj in <P2,...,Pn>, NextP mod Pj <> 0)
```

or

```
(for all Pj in <P1,...,Pn>, NextP mod Pj <> 0)
```

Thus whether or not `Primes` is empty, the desired function is computed. Since this design part meets its specification, the final comment can be refined.

Design Part 1.1.1.1.1.1

```
{(Primes = <P1,...,Pi> & <Pi+1,...,Pn> and Trial = Pi and i>=1 -->
     IsPrime := IsPrime and (NextP mod Trial <> 0) and
                (for all Pj in <Pi+1,...,Pn>, NextP mod Pj <> 0)) |
 (Trial = EndList --> )}
WHILE Trial <> EndList
DO
  BEGIN
    IsPrime := IsPrime AND (NextP mod Trial) <> 0;
    Next(Primes, Trial)
  END
```

To demonstrate that the WHILE statement computes the Design Part function, we make several simplifying assumptions. The only output variable is `IsPrime`; changes to `Trial` and `Primes` are ignored. We also discount the possibility of the result of the mod function being undefined in the case where the second argument is not a positive integer. Since these arguments are chosen from `Primes` whose members are all positive, the operation will always be defined.

The first two steps of the WHILE statement verification rule are straightforward. `Next(Primes,Trial)` assigns the value of the next element in the list `Primes` to `Trial` so eventually `EndList` must be returned. By assumption, the mod function is always defined. Thus the loop will terminate on the same domain for which the function is defined. The loop and the function both compute an identity function for the same domain. That is, when `Trial=EndList`, `Primes` is the empty list.

Let g be the specified function and T name the following program text:

```
IF Trial <> EndList
THEN
  BEGIN
    IsPrime := IsPrime AND (NextP mod Trial) <> 0;
    Next(Primes, Trial)
  END
```

The third condition of the WHILE statement verification rule is:

$g = \boxed{T} \circ g$.

Since there are two parts to the domain of Next, $\boxed{T}$ is

```
(Primes = <P1,...,Pk> & <Pk+1,...,Pn>
  and Trial = Pk and Trial ≠ EndList and k≥1 →
     Primes, Trial, IsPrime :=
          <P1,...,Pk+1> & <Pk+2,...,Pn>, Pk+1,
              IsPrime AND (NextP mod Trial) <> 0) |
(Primes = <P1,...,Pn> & <>
  and Trial = Pn and Trial ≠ EndList and n≥1 →
     Trial, IsPrime := EndList, IsPrime AND (NextP mod Trial) <> 0) |
(Trial = EndList → )
```

The following trace tables capture the composition of the first two parts of $\boxed{T}$ with g.

Case 1

Condition	IsPrime	Trial	Primes
Primes = <P1,...,Pk> & <Pk+1,...,Pn> and Trial = Pk and Trial ≠ EndList and k > 1	IsPrime and (NextP mod Trial ≠ 0)	Pk+1	<P1,...,Pk+1> & <Pk+2,...,Pn>
<P1,..,Pk+1> & <Pk+2,...,Pn> = <P1,...,Pi> & <Pi+1,...,Pn> and Pk+1 = Pi and i >= 1	IsPrime and (NextP mod Trial ≠ 0) and (NextP mod Pk+1 ≠ 0) and (for all Pj in <Pi+1,...,Pn>, NextP mod Pj ≠ 0)		

Setting i=k+1, we have

Condition:

```
Primes = <P1,...,Pi-1> & <Pi,...,Pn> and Trial = Pi-1 and
Trial ≠ EndList and k ≥ 1 and i ≥ 1 and i = k+1
```

Having eliminated k from the description of Primes and Trial, the last three parts of the condition can be replaced by i>=2. Trial=Pi-1 implies Trial≠EndList so the latter condition can also be omitted.

Assignment to IsPrime:

```
IsPrime and (NextP mod Trial ≠ 0) and (NextP mod Pi) ≠ 0) and
(for all Pj in <Pi+1,...,Pn>, NextP mod Pj <> 0)
```

The last two terms of the assignment combine to:

```
(for all Pj in <Pi,...,Pn>, NextP mod Pj <> 0)
```

The resulting conditional assignment is:

```
(Primes = <P1,...,Pi-1> & <Pi,...,Pn> and Trial = Pi-1 and i≥2 →
     IsPrime :=
          IsPrime and (NextP mod Trial ≠ 0) and
          (for all Pj in <Pi,...,Pn>, NextP mod Pj <> 0))
```

Composing the second and third parts of $\boxed{T}$ with the first part of g yields empty functions. The second part function of $\boxed{T}$ assigns the value `EndList` to `Trial` and the third part function equates these two values in its domain. Those states in which `Trial=EndList` are outside the domain of the first part of g.

The second part of $\boxed{T}$ is composed with the second part of g below.

Case 2

Condition	IsPrime	Trial	Primes
Primes = <P1,...,Pn> & <> and Trial = Pn and Trial ≠ EndList and n>=1	IsPrime and (NextP mod Trial ≠ 0)	EndList	
EndList = EndList			

Condition:

```
Primes = <P1,...,Pn> & <> and Trial=Pn and Trial≠EndList and n>=1
```

`Trial=Pn` implies `Trial≠EndList` so the latter condition can be omitted.

Assignment to `IsPrime`:

```
IsPrime and (NextP mod Trial ≠ 0)
```

Thus the conditional assignment is

```
Primes = <P1,...,Pn> & <> and Trial = Pn and n ≥ 1 →
     IsPrime :=
          IsPrime and (NextP mod Trial ≠ 0) and
          (for all Pj in <Pn+1,...,Pn>, NextP mod Pj <> 0)
```

since the last conjunct is vacuously true. The remaining compositions of the third part of $\boxed{T}$ with g result in an empty function and an identity function with domain guard `Trial=EndList`.

The function resulting from this composition is:

```
(Primes = <P1,...,Pi-1> & <Pi,...,Pn> and Trial = Pi-1 and i≥2 →
     IsPrime :=
          IsPrime and (NextP mod Trial ≠ 0) and
          (  Pj in <Pi,...,Pn>, NextP mod Pj ≠ 0) ) |
(Primes = <P1,...,Pn> & <> and Trial = Pn and n≥1 →
     IsPrime :=
          IsPrime and (NextP mod Trial ≠ 0) and
          (  Pj in <Pi+1,...,Pn>, NextP mod Pj ≠ 0) } |
(Trial = EndList → )
```

To convince ourselves that this function is the same as g, we observe the similarities between the guards of the first two parts of this function.

1. `Primes` is a list of integers separated into two parts, the first of which contains at least one element.
2. The value of `Trial` is the same as that of the last element in the first part of `Primes`, and that value differs from `EndList`.

Thus the second part of the function is just a special case of the first, and the two can be combined

yielding a function identical to *g*.

The final design parts initialize `Primes` and print its final values.

Design Part 1.2

```
{N>1 → Primes := <2>}
EmptyList(Primes);
Append(Primes,2);
```

Design Part 1.3

```
{print Primes}
Head(Primes,Trial);
WHILE Trial <> EndList
DO
  BEGIN
    WRITELN(Trial);
    Next(Primes,Trial)
  END
```

The assembled design parts are shown below.

```
PROGRAM PrintPrimes(OUTPUT);
  CONST
    N = 20;
    MaxSize = 20;
    EndList = 0;
  TYPE
    EltType = 0..MAXINT;
    List = ... ;
  VAR
    Primes: List;
    NextP: EltType;
    Trial: EltType;
    IsPrime: BOOLEAN;

  {include
  PROCEDURE EmptyList(VAR L: List);
    abs: L := <>

  PROCEDURE Append(VAR L: List; Elt: EltType);
    abs: (Length(L) < MaxSize --> L := L & <Elt>) |
         (Length(L) >= MaxSize --> )

  PROCEDURE Head(VAR L: List; VAR Result: INTEGER);
    abs: (L = <> --> Result := EndList) |
         (L = <L1,...,Li-1> & <Li,...,Ln> -->
              L, Result := <L1> & <L2,...,Ln>, L1)

  PROCEDURE Next(VAR L: List; VAR Result: INTEGER);
    abs: (L = <L1,...,Ln> & <> --> Result := EndList) |
         (L = <L1,...,Li-1> & <Li,...,Ln> -->
              L, Result := <L1,...,Li> & <Li+1,...,Ln>, Li) }
```

```
BEGIN {PrintPrimes}
  {N>1 → Primes := <2>}
  EmptyList(Primes);
  Append(Primes,2);
  NextP := 3;
  {3<=NextP<=N and Odd(NextP) -->
      Primes := Primes & OddPrimes(NextP, N)}
  WHILE NextP <= N
  DO
    BEGIN
       {IsPrime := (for all Pi in Primes, NextP mod Pi <> 0)};
       IsPrime := TRUE;
       Head(Primes,Trial);
       {(Primes = <P1,...,Pi> & <Pi+1,...,Pn>
          and Trial = Pi and i≥1 -->
             IsPrime := IsPrime and (NextP mod Trial <> 0) and
                 (for all Pj in <Pi+1,...,Pn>, NextP mod Pj <> 0)) |
        (Trial = EndList --> )}
       WHILE (Trial <> EndList) AND IsPrime
       DO
         BEGIN
           IsPrime := (NextP MOD Trial) <> 0;
           Next(Primes,Trial)
         END;
       IF IsPrime
       THEN
         Append(Primes,NextP);
       NextP := NextP + 2
    END;
  {print Primes}
  Head(Primes,Trial);
  WHILE Trial <> EndList
  DO
    BEGIN
      WRITELN(Trial);
      Next(Primes,Trial)
    END
END.  {PrintPrimes}
```

24.1.2 Verifying the Data Abstraction

The abstract type used to represent `Primes` is "list of integers with a reading pointer." Although `PrintPrimes` needs only a list of integers, we parametrize the `TYPE` declaration for `List` so that its size or the type of its elements could later change.

```
CONST
  MaxSize = 20;
  EndList = 0;
TYPE
  EltType = 1..MAXINT;
  List = RECORD
           Values: ARRAY [1..MaxSize] OF EltType;
           Size, Current: 0..MaxSize
         END;
```

The representation function for `List` is:

A_{List} = {(s,t): t is the same state as s except that in t `List` has value:
`<L.Values[1],...,L.Values[L.Current]> &`
`<L.Values[L.Current+1],...,L.Values[L.Size]>`
if `0<=L.Current<=L.Size<=MaxSize`; and t contains no members whose first elements are `L.Values`, `L.Current`, or `L.Size`}

The operations of this type are `EmptyList`, `Append`, `Head`, and `Next`. `InitList` is the special operation that produces `List` values.

```
PROCEDURE EmptyList(VAR L: List);
  {abs: L := <>
   con: L.Size, L.Current := 0, 0}
  BEGIN {EmptyList}
    L.Size := 0;
    L.Current := 0
  END;  {EmptyList}
```

The correspondence between the body of `EmptyList` and its concrete comment is apparent. Showing that the concrete comment is consistent with the abstract comment requires demonstrating that

$$EmptyList_{abs} \subseteq EmptyList_{con} \circ A_{List}.$$

That is,

```
<> = <L.Values[1],...,L.Values[0]>
       & <L.Values[1],...,L.Values[0]>
```

which verifies `EmptyList`.

`Head` is the special operation that carries `List` objects back to their `INTEGER` components in the concrete world, and `Next` maps `List` objects to `List` objects in the abstract world.

```
PROCEDURE Head(VAR L: List; VAR Result: EltType);
  { abs: (L = <> --> Result := EndList) |
         (L = <L1,...,Li-1> & <Li,...,Ln> -->
               L, Result := <L1> & <L2,...,Ln>, L1)
    con: (L.Size > 0 --> L.Current, Result := 1, L.Values[1]) |
         (L.Size <= 0 --> L.Current, Result := 1, EndList) }
  BEGIN {Head}
    IF L.Size > 0
    THEN
      BEGIN
        L.Current := 1;
        Result := L.Values[1]
      END
    ELSE
      Result := EndList
  END;  {Head}
```

```
PROCEDURE Next(VAR L: List; VAR Result: EltType);
  {abs: (L = <L1,...,Ln> & <> --> Result := EndList) |
        (L = <L1,...,Li-1> & <Li,...,Ln> -->
            L, Result := <L1,...,Li> & <Li+1,...,Ln>, Li)
   con: (1<=L.Current<L.Size -->
           L.Current,Result := L.Current+1,L.Values[L.Current+1]) |
        (L.Current < 1 or L.Current <= Size --> Result := EndList)}
  BEGIN {Next}
    IF (1 <= L.Current) AND (L.Current < L.Size)
    THEN
      BEGIN
        L.Current := L.Current + 1;
        Result := L.Values[L.Current]
      END
    ELSE
      Result := EndList
  END;  {Next}
```

Again the correspondence between the bodies of `Head` and `Next` and their concrete comments are clear. Correctness of the implementations requires:

$$A_{List} \circ Head_{abs} \subseteq Head_{con} \circ A_{List}$$

and

$$A_{List} \circ Next_{abs} \subseteq Next_{con} \circ A_{List}.$$

Only the first proof will be given.

$A_{List} \circ Head_{abs}$

Condition	L	Result
0≤L.C≤L.S≤ MaxSize	<L.V[1],...,L.V[L.C]> & <L.V[L.C+1],...,L.V[L.S]>	
<L.V[1],...,L.V[L.C]> & <L.V[L.C+1],...,L.V[L.S]> = <>		EndList

The condition evaluates to TRUE when

```
<L.V[1],...,L.V[L.C]> = <> AND
   <L.V[L.C+1],...,L.V[L.S]> = <>
```

Picking `L.S` and `L.C` to be 0 achieves this result. Thus the function is

```
MaxSize>=0 AND L.S=0 AND L.C=0 → L,Result := <>,EndList
```

$A_{List} \circ Head_{abs}$

Condition	L	Result
0≤L.C≤L.S≤ MaxSize	<L.V[1],...,L.V[L.C]> & <L.V[L.C+1],...,L.V[L.S]>	
<L.V[1],...,L.V[L.C]> & <L.V[L.C+1],...,L.V[L.S]> = <L1,...,Li-1> & <Li,...,Ln>	<L1> & <L2,...,Ln>	L1

This condition is TRUE when i≥1 (i.e., L.S≥L.C≥1) and L.V[1]=L1, ..., L.V[L.S]=Ln. Thus the function is

```
(MaxSize≥L.S≥L.C≥1 → L, Result := <L1> & <L2,...,Ln>, L1
```

The composition $A_{List} \circ Head_{abs}$ yields

```
(MaxSize>=0 AND L.S=0 AND L.C=0 → L,Result := <>,EndList) |
(MaxSize>=L.S>=L.C>=1 → L,Result := <L1> & <L2,...,Ln>, L1)
```

$Head_{con} \circ A_{List}$

Condition	L	L.C	Result
L.S>0		1	L.V[1]
0≤1≤L.S≤ MaxSize	<L.V[1],...,L.V[L.C]> & <L.V[L.C+1],...,L.V[L.S]>		

Thus the function is

```
(MaxSize>=L.S>=1 →
     L,Result := <L.V[1]> & <L.V[2],...,L.V[L.S]>, L.V[1])
```

$Head_{con} \circ A_{List}$

Condition	L	L.C	Result
L.S<=0			EndList
0≤L.C≤L.S≤ MaxSize	<L.V[1],...,L.V[L.C]> & <L.V[L.C+1],...,L.V[L.S]>		

Since the condition requires `L.S<=0` and `L.S>=0`, it must be the case that `L.S=0`, and hence `L.C=0`, so the list

```
<L.V[1],...,L.V[L.C]> & <L.V[L.C+1],...,L.V[L.S]>
```

must be empty. Thus the function is

```
MaxSize>=0 AND L.S=0 AND L.C=0 → L,Result := <>,EndList
```

The composition $Head_{con} \circ A_{List}$ yields

```
(MaxSize>=L.S>=1 →
   L,Result := <L.V[1]> & <L.V[2],...,L.V[L.S]>, L.V[1]) |
(MaxSize>=0 AND L.S=0 AND L.C=0 → L,Result := <>,EndList)
```

This composition yields the same results as the previous composition, but on a slightly larger domain since it places no restriction on the initial value of `L.C`.

Append maps a `List` and an `EltType` value to a `List` value.

```
PROCEDURE Append(VAR L: List; Elt: EltType);
  {abs: (Length(L) < MaxSize --> L := L & <Elt>) |
        (Length(L) >= MaxSize --> )
   con: (L.Size < MaxSize --> L.Values, L.Size :=
            alpha(L.Values,L.Size+1,Elt), L.Size + 1) |
        (L.Size >= MaxSize --> ) }
  BEGIN {Append}
    IF L.Size < MaxSize
    THEN
      BEGIN
        L.Size := L.Size + 1;
        L.Values[L.Size] := Elt
      END
  END;  {Append}
```

24.1.3 Integrating and Testing the Design Parts

The test plan for `PrintPrimes` should cover not only the program but the abstract data type `List` defined within it. The data abstraction is potentially reusable, so its testing will be independent of `PrintPrimes`.

To test the data abstraction functionally, we should consider the possible logical sequences of the operations: `EmptyList`, `Append`, `Head`, and `Next`. For example, to check valid combinations of operations we might:

1. build a list,
2. sequence through a list,
3. go back to the head of the list after partially sequencing through the list (after a `Head` or a `Next`),
4. append an element to the list while sequencing through it (after a `Head` or a `Next`),
5. empty the list after sequencing through or building it.

These operations should be considered with at least three different kinds of list values: an empty list, a partially full list, and a full list (since we have chosen an array implementation). Some other combinations of operations are not legal and are not tested. For example, applying the `Next` operation to a list without executing the `Head` operation first will cause an error.

To check that all cases are covered, consider all combinations of operations, noting whether they are legal or illegal combinations and whether the combination can appear in the program `PrintPrimes`. We must at least cover those that appear in `PrintPrimes`.

Case	Operations	Legality	`PrintPrimes`?
1	`EmptyList;EmptyList`	legal	no
2	`EmptyList;Append`	legal	yes
3	`EmptyList;Head`	legal	no
4	`EmptyList;Next`	illegal	no
5	`Append;EmptyList`	legal	no
6	`Append;Append`	legal	no
7	`Append;Head`	legal	yes
8	`Append;Next`	illegal	no
9	`Head;EmptyList`	legal	no
10	`Head;Append`	legal	no
11	`Head;Head`	legal	no
12	`Head;Next`	legal	yes
13	`Next;EmptyList`	legal	no
14	`Next;Append`	legal	no
15	`Next;Head`	legal	yes
16	`Next;Next`	legal	yes

To test the data abstraction properly so that it can be reused, we should at least check the possible combinations of operations listed above on three representative list values: an empty list, a partially full list, and a full list. To make testing easier, the maximum list size is temporarily restricted to four elements.

```
PROGRAM TestList(OUTPUT);
  CONST
    MaxSize = 4; {short lists for now}
    EndList = 0;
  TYPE
    EltType = 0..MAXINT;
    List = ... ;
  VAR
    L: List;
    NextP: EltType;
    Trial: EltType;
  {include
  PROCEDURE EmptyList(VAR L: List);
    abs: L := <>
  PROCEDURE Append(VAR L: List; Elt: EltType);
    abs: (Length(L) < MaxSize --> L := L & <Elt>) |
         (Length(L) >= MaxSize --> )
  PROCEDURE Head(VAR L: List; VAR Result: INTEGER);
    abs: (L = <> --> Result := EndList) |
         (L = <L1,...,Li-1> & <Li,...,Ln> -->
              L, Result := <L1> & <L2,...,Ln>, L1)
  PROCEDURE Next(VAR L: List; VAR Result: INTEGER);
    abs: (L = <L1,...,Ln> & <> --> Result := EndList) |
         (L = <L1,...,Li-1> & <Li,...,Ln> -->
              L, Result := <L1,...,Li> & <Li+1,...,Ln>, Li)
  PROCEDURE Test(VAR L: List); }
  BEGIN {TestList}
    EmptyList(L);
    WRITELN('testing <>');
    Test(L);
    EmptyList(L);
    Append(L,1); {case 2 EmptyList; Append}
    Append(L,2); {case 6 Append; Append}
    WRITELN('testing <1,2>');
    Test(L);
    EmptyList(L);
    Append(L,1);
    Append(L,2);
    Append(L,3);
    WRITELN('testing <1,2,3>');
    Test(L)
  END.  {TestList}
```

Procedure `Test` shown below covers most legal combinations of operations except those ending with `EmptyList`. Note that there are two different methods used to write list elements: sequencing through the list using `Next` and printing the array elements directly. We do this so that the list elements can be displayed without changing the last operation executed.

```
PROCEDURE Test(VAR L: List);
  VAR
    Temp: EltType;
    I: INTEGER;
  BEGIN {Test}
    WRITE('enter test, l= <');
    FOR I := 1 to L.Size
    DO
      WRITE(L.Values[I]: 4)
    WRITELN(' >');
    Head(L,Temp);
    WRITELN('(emptylist or append); head =', Temp);
    Head(L,Temp);
    WRITELN('head; head =', Temp);
    WRITE('full sequence: head; next**i, i>=0: <');
    WHILE Temp <> EndList
    DO
      BEGIN
        WRITE(Temp: 4);
        Next(L,Temp)
      END;
    WRITELN(' >');
    Head(L,Temp);
    WRITELN('next**i; head =', Temp);
    IF Temp <> EndList
    THEN
      BEGIN
        Next(L,Temp);
        WRITELN('part sequence: head; next =', Temp)
      END;
    Head(L,Temp);
    WRITELN('head;(next or null);head =', Temp);
    WRITE('head; append(15) = <');
    Append(L,15);
    FOR I := 1 to L.Size
    DO {write the list elements without Head or Next}
      WRITE(L.Values[I]: 4)
    WRITELN(' >');
    Head(L,Temp);
    IF Temp <> EndList
    THEN
      BEGIN
        WRITE('next; append(25) = <');
        Next(L,Temp);
        Append(L,25);
        FOR I := 1 to L.Size
        DO
          WRITE(L.Values[I]: 4)
        WRITELN(' >')
      END;
    Head(L,Temp);
    WRITELN('append; head =', Temp);
    WRITE('full sequence: head; next**i, i>=0: <');
    WHILE Temp <> EndList
```

```
    DO
      BEGIN
        WRITE(Temp: 4);
        Next(L,Temp)
      END;
    WRITELN(' >');
    WRITELN
  END;  {Test}
```

Sample output for the three tests and the cases covered are shown below.

Test Results

Output	Case
testing <>	
enter test, l= < >	
(emptylist or append); head = 0	3
head; head = 0	11
full sequence: head; next**i, i>=0: < >	
next**i; head = 0	11
head;(next or null);head = 0	11
head; append(15) = < 15 >	10
next; append(25) = < 15 25 >	14
append; head = 15	7
full sequence: head; next**i, i>=0: < 15 25 >	12,16
testing <1,2>	
enter test, l= < 1 2 >	
(emptylist or append); head = 1	7
head; head = 1	11
full sequence: head; next**i, i>=0: < 1 2 >	12,16
next**i; head = 1	15
part sequence: head; next = 2	12
head;(next or null);head = 1	12,15
head; append(15) = < 1 2 15 >	10
next; append(25) = < 1 2 15 25 >	8,14
append; head = 1	7
full sequence: head; next**i, i>=0: < 1 2 15 25 >	12,16
testing <1,2,3>	
enter test, l= < 1 2 3 >	
(emptylist or append); head = 1	7
head; head = 1	11
full sequence: head; next**i, i>=0: < 1 2 3 >	12,16
next**i; head = 1	15
part sequence: head; next = 2	12
head;(next or null);head = 1	12,15
head; append(15) = < 1 2 3 15 >	10
next; append(25) = < 1 2 3 15 >	8,14
append; head = 1	7
full sequence: head; next**i, i>=0: < 1 2 3 15 >	12,16

We act as test oracles judging that each Head or Next operation returns the correct result. We also note that appending two elements to an empty list yields a 2-list, to a 3-list yields a 4-list (the last Append is ignored), and to a 4-list leaves the list unchanged. Our test program fails to cover several simple cases. Cases 2 and 6 are covered in the body of the test program when the list objects are constructed. Cases 1, 5, 9, and 13 all concern setting an object to the EmptyList after other

operations have been performed on it, and is best left for another test program.

Now consider the test data from the point of view of some structural coverage criterion. Suppose the criterion was to check whether the tests execute every statement in the `List` operations. These tests satisfy it because the `List` operations have simple implementations and the criterion is weak. To satisfy a more realistic criterion would require more combinations of operations.

`PrintPrimes` is a small program and its design parts may be integrated in two development programs. Design Parts 1-1.1.1.1, 1.2, and 1.3 could be integrated at once and Design Part 1.1.1.1.1

```
{IsPrime := (For all members P of Primes, NextP mod P <> 0)}
```

could be replaced with the assignment statement

```
IsPrime := TRUE;
```

to produce the first development program that printed a list containing 2 followed by all the odd numbers less than or equal to `N`. The second development program would integrate the final two design parts needed to produce the appropriate value for `IsPrime`.

To test the program `PrintPrimes` structurally, we should pick interesting values for `N` such that `N>=1`. Thus we should set `N` to its minimum value (2), the value of the `MaxSize`th (i.e., 20^{th}) prime (71), some intermediate prime (e.g., 29) and non-prime (e.g., 30) values, and a value greater than the `MaxSize`th prime that includes other primes (e.g., 100). Any value of `N` greater than 2 will cause each of the statements in the body of `PrintPrimes` to be executed. Since 3 is prime, it will exercise both inner WHILE statements and the `Append`.

24.2 Chapter Summary

In this chapter, we have summarized many of the program development and analysis techniques presented in the text. Both stepwise refinement and data abstraction were used to produce the solution. Analysis of the program proceeded in parallel with the development to make certain that each design part computed a function that was consistent with its specification. In trying to make the analysis task "scale up" to more complex problems, we ignored variables that were used for purely local computation and dealt with a data type (`List`) with nice mathematical properties instead of manipulating arrays directly (which would have added unnecessary detail to the proof).

Testing proceeded in two stages: the data abstraction was tested alone, and then the design parts of the program were combined into development programs so that intermediate results could be checked. Both functional and structural testing criteria were used to select test data and evaluate how thoroughly the data exercised the pieces of the program.

Appendix I: Collected Syntax for CF Pascal

Syntax Rules

A string is a CF Pascal program (subject to the restrictions on standard words given in Appendix II and the context rules given below) if and only it it can be obtained by reading the leaves from a syntax tree rooted at <program> from left to right and inserting blanks and comments appropriately. The syntax trees for CF Pascal are constructed using the following rules.

SR1. <program> ::= <heading> ; <block> .

SR2. <heading> ::= PROGRAM <identifier> (INPUT, OUTPUT)

SR3. <identifier> ::= <letter>
| <identifier> <letter>
| <identifier> <digit>

SR4. <letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n |
o | p | q | r | s | t | u | v | w | x | y | z

SR5. <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

SR6. <block> ::= <BEGIN statement>
| <declarations> ; <BEGIN statement>
| <procedure list> ; <BEGIN statement>
| <declarations> ; <procedure list> ; <BEGIN statement>

SR7. <declarations> ::= VAR <identifier list> : <type>
| <declarations> ; <identifier list> : <type>

SR8. <identifier list> ::= <identifier>
| <identifier list> , <identifier>

SR9. <BEGIN statement> ::= BEGIN <statement list> END

SR10. <statement list> ::= <statement>
| <statement list> ; <statement>

SR11. <statement> ::= <READ statement>
| <WRITE statement>
| <assignment statement>
| <null statement>
| <BEGIN statement>
| <IF statement>
| <WHILE statement>
| <procedure statement>

SR12. <READ statement> ::= READ (<identifier list>)
| RESET (<identifier>) | READLN (<identifier list>)

SR13. <WRITE statement> ::= WRITE (<write list>)
| WRITELN (<write list>)
| WRITELN
| REWRITE (<identifier>)

SR14. <write list> ::= <write item>
| <write list> , <write item>

SR15. <write item> ::= <identifier>
| ' <character string> '

SR16. <character string> ::= <character>
| <character string> <character>

SR17. <character> ::= <letter> | <digit> | <special character>

SR18. <special character> ::= (|) | = | { | } | : | ; | '
| □ | < | > | , | . | #

SR19. <assignment statement> ::= <identifier> := <expression>

SR20. <expression> ::= <identifier>
| ' <character> '

SR21. <null statement> ::=

SR22. <IF statement> ::= IF <condition> THEN <statement>
| IF <condition> THEN <statement> ELSE <statement>

SR23. <condition> ::= <expression> <comparison> <expression>
| NOT (<condition>)
| (<condition>) AND (<condition>)
| (<condition>) OR (<condition>)
| EOF | EOF (<identifier>) | EOLN | EOLN (<identifier>)

SR24. <comparison> ::= < | = | > | <= | <> | >=

SR25. <WHILE statement> ::= WHILE <condition> DO <statement>

SR26. <procedure list> ::= <procedure>
| <procedure list> ; <procedure>

SR27. <procedure> ::= <procedure heading> ; <block>

SR28. <procedure statement> ::= <identifier> |
<identifier> (<actual parameter list>)

SR29. <type> ::= CHAR | TEXT

SR30. <actual parameter list> ::= <identifier list>

SR31. <procedure heading> ::= PROCEDURE <identifier> |
PROCEDURE <identifier> (<formal parameter list>)

SR32. <formal parameter list> ::= VAR <identifier list> : <type> |
<formal parameter list> ; VAR <identifier list> : <type>

Context Rules

In addition to the syntax rules, CF Pascal programs must satisfy the following context rules.

CR1. All identifiers in a declaration's identifier list must be unique.

CR2. No reserved word, e.g. IF, WHILE, etc. can be used as an identifier. (A list of the reserved words in Pascal is given at the end of the next Section.)

CR3. Any <identifier> in a <statement> that is not a <procedure statement>, other than standard identifiers (e.g., READ, WRITE, etc.) must appear in the <identifier list> of the <declarations>.

CR4. The procedure identifier used in a <procedure statement> must appear in a <procedure heading> of a <procedure> declaration. The <actual parameter list> of a <procedure statement> must agree in length with the <formal parameter list> of the associated <procedure heading>. The identifiers in the <formal parameter list> and the <actual parameter list> must agree in type.

CR5. Variable and procedure names can't be the same <identifier>.

CR6. A TEXT identifier is an <identifier> declared to be of <type> TEXT. If a TEXT identifier appears in a <READ statement> or a <WRITE statement>, it must be the first <identifier>. A TEXT identifier, other than INPUT or OUTPUT, that appears in a READ statement or WRITE statement must appear in a <declarations>. Only TEXT identifiers other than INPUT and OUTPUT may appear in RESET or REWRITE statements; only TEXT identifiers that have appeared in a RESET statement may appear in EOF or EOLN <condition>s. Any identifier declared in a procedure's <formal parameter list> may only be referenced in the <BEGIN statement> of the procedure.

CR7. An identifier may not be used outside its scope.

Appendix II: STANDARD PASCAL WORDS

Reserved Words

Although the standard Pascal word PROGRAM appears to be an <identifier>, it may not be used as one in forming Pascal programs. PROGRAM is a *reserved word* in Pascal, and its use is restricted to the places it appears literally in a syntax rule (only in a <heading>). The reserved words of Pascal are listed in the following table. None of these words may be used as an <identifier> in CF Pascal or in full Pascal.

Reserved Words in Pascal

AND	END	NIL	SET
ARRAY	FILE	NOT	THEN
CASE	FUNCTION	OR	TYPE
CONST	GOTO	PACKED	UNTIL
DIV	IF	PROCEDURE	VAR
DO	IN	PROGRAM	WHILE
DOWNTO	LABEL	RECORD	WITH
ELSE	MOD	REPEAT	

Standard Identifiers

Some standard Pascal words (for example, INPUT, OUTPUT, WRITE) are not reserved, but nevertheless should be avoided. They are *standard identifiers* that are by default attached to some special purpose. The standard identifiers of Pascal are:

Standard Identifiers in Pascal

ABS	GET	PAGE	SQR
ARCTAN	INPUT	PRED	SQRT
BOOLEAN	INTEGER	PUT	SUCC
CHAR	LN	READ	TEXT
CHR	MAXINT	READLN	TRUE
COS	NEW	REAL	TRUNC
EOF	ODD	RESET	UNPACK
EOLN	ORD	REWRITE	WRITE
EXP	OUTPUT	ROUND	WRITELN
FALSE	PACK	SIN	

Appendix III: Syntax of Pascal

In alphabetical order by left side:

<actual parameter> ::= <expression> | <procedure identifier> | <function identifier>
<actual parameter list> ::= <actual parameter list> , <actual parameter>
| <actual parameter>
<actual parameter list> ::= (<actual parameter list>) |
<add operator> ::= + | - | OR
<array variable> ::= <variable access>
<assignment statement> ::= <identifier> := <expression>

<base type> ::= <ordinal type>
<BEGIN statement> ::= BEGIN <statement list> END
<block> ::= <label declaration part> <constant definitions>
<type definition part> <variable declarations>
<proc/func declarations> <BEGIN statement>
<buffer variable> ::= <file variable> ↑

<case> ::= <constant list> : <statement>
<case list> ::= <case list> ; <case> | <case>
<case statement> ::= CASE <expression> OF <case list> END
<character> ::= <letter> | <digit> | <special character>
<character string> ::= <character> | <character string> <character>
<comparison> ::= < | = | > | <= | <> | >=
<component type> ::= <type denoter>
<component variable> ::= <field designator> | <indexed variable>
<condition> ::= <expression> <comparison> <expression> | NOT (<condition>)
| (<condition>) AND (<condition>) | (<condition>) OR (<condition>)
| EOF | EOF (<identifier>) | EOLN | EOLN (<identifier>)
<constant> ::= <sign> <unsigned number> | <unsigned number>
| <sign> <constant identifier> | <constant identifier> | <character string>
<constant definition> ::= <identifier> = <constant> ;
<constant definition part> ::= CONST <constant definitions> |
<constant definitions> ::= <constant definitions> <constant definition> | <constant definition>
<constant identifier> ::= <identifier>
<constant list> ::= <constant list> , <constant> | <constant>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digit sequence> ::= <digit sequence> <digit> | <digit>

<enumerated type> ::= (<identifier list>)
<expression> ::= <simple expression>
| <simple expression> <relational operator> <simple expression>
<expression list> ::= <expression list> , <expression> | <expression>

<factor> ::= <variable access> | <function designator> | <unsigned constant>
| (<expression>) | NOT <factor> | <set constructor>
<field designator> ::= <record variable> . <field specifier> | <field designator identifier>
<field designator identifier> ::= <identifier>
<field list> ::= <fixed part> <variant part> | <fixed part> | <variant part> |
<field specifier> ::= <identifier>

<file variable> ::= <variable access>
<fixed part> ::= <record sections>
<for list> ::= <expression> TO <expression> | <expression> DOWNTO <expression>
<for statement> ::= FOR <variable identifier> := <for list> DO <statement>
<formal parameter> ::= <value parameter> | <variable parameter> | <procedural parameter>
<formal parameter list> ::= (<formal parameters>) |
<formal parameters> ::= <formal parameters> ; <formal parameter> | <formal parameter>
<function designator> ::= <function identifier> <actual parameter list>
<function identifier> ::= <identifier>

<goto statement> ::= GOTO <label>

<heading> ::= PROCEDURE <identifier> <formal parameter list>
  | FUNCTION <identifier> <formal parameter list> : <result type>

<identified variable> ::= <pointer variable> ↑
<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>
<identifier list> ::= <identifier> | <identifier list> , <identifier>
<IF statement> ::= IF <condition> THEN <statement>
  | IF <condition> THEN <statement> ELSE <statement>
<index type list> ::= <index type list> , <ordinal type> | <ordinal type>
<indexed variable> ::= <array variable> [<expression list>]

<label> ::= <digit sequence>
<label declaration part> ::= LABEL <label list> ; |
<label list> ::= <label list> , <label> | <label>
<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
  | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<member designator> ::= <expression> | <expression> .. <expression>
<member designators> ::= <member designators> , <member designator> |
<mult operator> ::= * | DIV | MOD | AND

<new ordinal type> ::= <enumerated type> | <subrange type>
<new pointer type> ::= ↑ <type identifier>
<new structured type> ::= <unpacked structured type>
  | PACKED <unpacked structured type>
<new type> ::= <new ordinal type> | <new structured type> | <new pointer type>
<null statement> ::=

<optional tag field> ::= <identifier> : |
<ordinal type> ::= <new ordinal type> | <ordinal type identifier>
<ordinal type identifier> ::= <type identifier>

<pointer variable> ::= <variable access>
<procedural parameter> ::= <heading>
<procedure statement> ::= <identifier> | <identifier> (<actual parameter list>)
<proc/func declaration> ::= <heading> ; <block> ;
<proc/func declarations> ::= <proc/func declaration>
  | <proc/func declarations> <proc/func declaration>
<program> ::= <program heading> ; <block> .
<program heading> ::= PROGRAM <identifier> (<identifier list>)

<READ statement> ::= READ (<identifier list>)

¦RESET (<identifier>) ¦READLN (<identifier list>)

<record section> ::= <identifier list> : <type denoter>

<record sections> ::= <record sections> ; <record section> ¦ <record section>

<record variable> ::= <variable access>

<record variable list> ::= <record variable list> , <record variable> ¦ <record variable>

<relational operator> ::= = ¦ <> ¦ < ¦ <= ¦ > ¦ >= ¦ IN

<repeat statement> ::= REPEAT <statement list> UNTIL <expression>

<repetitive statement> ::= <WHILE statement> ¦ <repeat statement> ¦ <for statement>

<result type> ::= <simple type identifier> ¦ <pointer type identifier>

<set constructor> ::= [<member designators>]

<sign> ::= + ¦ -

<signed integer> ::= <sign> <unsigned integer> ¦ <unsigned integer>

<simple expression> ::= <simple expression> <add operator> <term> ¦ <term> ¦ <sign> <term>

<simple statement> ::= <assignment statement> ¦ <procedure statement> ¦ <READ statement>
¦ <WRITE statement> ¦ <null statement> ¦ <goto statement> ¦ <case statement>

<special character> ::= (¦) ¦ = ¦ { ¦ } ¦ : ¦ ; ¦ ' ¦ □ ¦ < ¦ > ¦ , ¦ . ¦ #

<statement> ::= <unlabeled statement> ¦ <label> : <unlabeled statement>

<statement list> ::= <statement list> ; <statement> ¦ <statement>

<structured statement> ::= <BEGIN statement> ¦ <IF statement>
¦ <repetitive statement> ¦ <with statement>

<subrange type> ::= <constant> . . <constant>

<term> ::= <term> <mult operator> <factor> ¦ <factor>

<type definition> ::= <identifier> = <type denoter> ;

<type definition part> ::= TYPE <type definitions> ¦

<type definitions> ::= <type definitions> <type definition> ¦ <type definition>

<type denoter> ::= <type identifier> ¦ <new type>

<type identifier> ::= <identifier>

<unlabeled statement> ::= <simple statement> ¦ <structured statement>

<unpacked structured type> ::= SET OF <base type> ¦ FILE OF <component type>
¦ RECORD <field list> END ¦ ARRAY [<index type list>] OF <component type>

<unsigned constant> ::= <unsigned number> ¦ <constant identifier> ¦ <character string>

<unsigned integer> ::= <digit sequence>

<unsigned number> ::= <unsigned integer> ¦ <unsigned real>

<unsigned real> ::= <unsigned integer> . <digit sequence>
¦ <unsigned integer> . <digit sequence> E <signed integer>
¦ <unsigned integer> E <signed integer>

<value parameter> ::= <identifier list> : <type identifier>

<variable access> ::= <variable identifier> ¦ <buffer variable>
¦ <component variable> ¦ <identified variable>

<variable declaration> ::= <identifier list> : <type denoter>

<variable declaration part> ::= VAR <variable declarations> ; ¦

<variable declarations> ::= <variable declarations> <variable declaration>
¦ <variable declaration>

<variable parameter> ::= VAR <identifier list> : <type identifier>

<variant> ::= <constant list> : (<field list>)

<variant list> ::= <variant list> ; <variant> ¦ <variant>

<variant part> ::= CASE <variant selector> OF <variant list>

<variant selector> ::= <optional tag field> <ordinal type identifier>

<WHILE statement> ::= WHILE <condition> DO <statement>
<with statement> ::= WITH <record variable list> DO <statement>
<write item> ::= <identifier> | ' <character string> '
<write item list> ::= <file variable> <write parameters> | <write parameters>
<write list> ::= <write item> | <write list> , <write item>
<write parameter> ::= <expression> : <expression> | <expression>
<write parameter list> ::= (<write item list>)
<write parameters> ::= <write parameters> <write parameter> | <write parameter>
<WRITE statement> ::= WRITE (<write list>)
| WRITELN (<write list>) | WRITELN | REWRITE (<identifier>)

INDEX

ABS (integer or real function) 429, 595
abstract
  data objects 349, 483, 485, 569
  operation 350, 485
  operation correctness 500
alias 246
AND (Boolean operator) 101
  meaning 193
annotated output 21
appearance rule 82
ARCTAN (real function) 595
arithmetic operators
  (+ - * / DIV MOD) 386, 594
ARRAY data type 513
  array-value relation 518
  analysis of programs 543
  declaration meaning 518
  dimension 538
  expression meaning 518
  operations 513
  PACKED attribute 533
  parameters 519
  representing functions 513
  representing other types 521, 549
  searching and sorting 525
assignment rule 188
assignment statement 20
  accumulating 406
  array 513, 543
  meaning 173, 401, 519, 543
associativity 104, 381
authentication 372

BEGIN statement 8, 26, 184
  design rule 187, 403
  meaning 175
bisection 620
block 51
  meaning 170
BNF (Backus-Naur Form) 46, 273, 334
BOOLEAN
  condition rule 231
  conditions 23, 99, 105, 192
  constants 192, 379
  data type 379
  expressions 102
  expression meaning 192, 383
  identities 103
  input/output 384
  operators 99, 379
  values 99, 379
box notation 160
buffer variable 442

canonical form 489
CASE statement 467
  meaning 470
CHAR data type 15
  constants 20, 143
CHR (character function) 430
closed statement 561
collisions 533, 631
comment 15, 62
  abstract 489
  concrete 489
  include 255
commutativity
  operators 104
  diagrams 501
compiler 7
composition
  function, relation (o) 166, 222
  power notation (r^n) 216
composition
  list, string ($\bigtriangledown$) 145, 149
concatenation
  list, string (&) 144, 149
concordance
concrete
  data 349, 484
  part function 350
concurrent assignment 180
conditional assignment 197, 339
  comparison 204
  composition (o) 222
conditions (see Boolean)
CONST declarations 415, 613
context rules 51
control variable 472
copy rule 82
correctness of module 500
Correctness Theorem 336
COS (real function) 595
cursor 16

dangling reference 561
data type 15, 372
  abstract 342, 483, 549, 569

aggregate 373, 436
ordinal 373, 594
simple 373
declaration 15
array 518
constant 415
extent (§) 252
meaning of VAR 168
meaning of procedure 170, 325
meaning of file 443
scope 252, 461, 620
type 373, 417
deMorgan's laws 104
dereference operation
(↑ on pointer value) 560
design part 68
development program 68
DISPOSE (pointer operation) 560, 630
distributivity 104
DIV (integer operator) 386-387
division (/) (real operator) 594
DO Part rule 231
domain of a function or relation 157
domain of an array 513

echoed input 21
encapsulation 344
enumerated data type 373
meaning 375
input/output 375
EOF 131, 390, 456
meaning 192, 444
EOLN 130, 456
meaning 192
equality symbol (=)
(see relational operators)
execution
aborted 20
conditional 23, 92
endless 32
iterative 30
normal 20
sequential 10, 163
symbolic 182
state 164, 561
table 11
existence rule 231, 404
EXP (real function) 595
exponent 593
exponentiation 595

factorial 409
Fibonacci number 409
field 457
tag 628
file data type 442, 521
operations 443
meaning of declaration 169, 443
TEXT (see TEXT file)
finite state machine 352, 443
floating point notation 593
FOR statement 471
meaning 474
analysis 477
format directives 389, 595
FORWARD declarations 433
function 157, 370
domain 157
composition (o) 166
constant 157
empty ({}) 157
identity (I) 157
inverse 168, 403
power notation (r^n) 216
range 157
transpose (r^T) 168
undefined 157
value notation 157
zero of a function 620
FUNCTION
declaration 423, 569
meaning 424
parameter 616
predefined 426

garbage 563
GCD 492, 494, 583
(greatest common divisor)
GET statement
meaning 444
global references 251, 256, 619
GOTO statement 573
and compound statements 579
guarded assignment (→) 198

harmonic series 486, 570
hash
code 532
function 631
table 631
head of a string or list (Θ) 146, 149
Horner's rule 397, 608
host type 401

identifier 8
identity function 157, 199
IF statement 25, 28
  design rule 210
  meaning 195
implementation diagrams 501
IN (set membership operator) 436
include comment 255
induction 321
information hiding 344
INTEGER data type 386
  constants 386
  expression meaning 396
  operators 386
  input/output 389
inverse of function, relation 168, 403

label 573
lazy evaluation 379, 455
lexicographic order 254, 535
line marker (/) 11, 130
linked structures 548, 564
list 148, 548, 564
  composition (∇) 149
  concatenation (&) 149
  empty (<>) 149
  membership ($\in$) 148
  sublists 149
LN (real function) 595
local references 243, 249, 256
locator value (of pointer) 559

machine
  Pascal 7
  state 342, 351
mantissa 593
matrix 538
  product 539
MAXINT 386
membership in list, set ($\in$) 148, 154
MOD (integer operator) 387
module 342, 344, 483

NEW (pointer operation) 560, 630
NIL (pointer constant) 560
nonlocal references 251, 256, 619
NOT (Boolean operator) 100
  meaning 193
not equal symbol (<>)
  (see relational operators)
null statement 27, 199
  meaning 175

number
  base 261, 397
  conversion 397
  natural 261

observable values 609
ODD (integer function) 429
open statement 561
operators
  associative 104, 381, 387
  binary 100
  commutative 104
  distribution 104
  infix 100
  precedence 382, 387
  prefix 99
  unary 99
OR (Boolean operator) 101
  meaning 193
ORD (integer function) 430
overflow 392, 403, 492

PACK (array operation) 533
PAGE statement 448
parameters 241, 418, 420
  actual 242
  array 519
  formal 242
  FUNCTION 616
  PROCEDURE 616
  value 420, 461, 535
  value parameter meaning 421
  VAR 242, 420
  VAR parameter meaning 245
perfect results (real operation) 608
period meaning 166
pointer data type 559
  locator value 559
  operations 560
  representing other types 569
  target value 560
  to variant records 630
precedes symbol (<) 23
  (see relational operators)
PRED (ordinal function) 426
preservation rule 188, 403
prime numbers 439
procedure declaration 78, 241
  meaning 170
procedure parameter 616
procedure statement 79, 241
  meaning 179, 258, 325, 421

recursive verification rule 326, 331
program 7
analysis 404
correctness 335, 500
libraries 239, 255, 269
meaning 142, 159
specification 239, 333, 339
structured 560
unstructured 560
program header 8
meaning 165
proof 337
by contradiction 234
by induction 321
PUT statement
meaning 444

quasi-random number 599
question mark (?) 16
queue
FIFO 354
priority 370, 598

range of a function or relation 157
rational numbers 486, 570
READ statement 16, 19, 110, 447
meaning 177
READLN statement 131
meaning 177
REAL data type 593
constants 593
errors analysis
input/output 595
operations 594
RECORD data type 457, 483
meaning 459
variant part 627
recurrence equation 221, 326
recursion 239, 308
in definition 47, 51, 382
mutual 433
tail 313
relation 156, 370
composition (o) 166
domain 157
inverse 168
power notation (r^n) 216
range 157
transpose (r^T) 168
relational operators
(< > <= >= = <>)
24, 25, 192-194, 594
relative errors (in real expressions) 608
REPEAT statement 480
meaning 481
replication 372
representation function 349, 484, 502
representational errors
(for real values) 607
reserved word 51
RESET statement 110, 448
meaning 177, 444
REWRITE statement 110
ROUND (real function) 594
meaning 176, 443
roundoff errors (in real expressions) 606

scaling factor 593
scientific notation 593
scope of identifiers 252, 461, 620
search 525
binary 526
depth-first 559, 567
hash 532, 631
sequential 526
sentinel 391, 526
set 153, 370
builder notation 154
constant 153, 436
data type 435
difference (-) 155, 436
empty ({}) 154
expression meaning 438
intersection ($\cap$ or *) 155, 436
membership ($\in$ or IN) 154, 436
power set 155, 436
singleton 154
subset ($\subseteq$ or <=) 154, 436
superset (>=) 436
union ($\cup$ or +) 155, 436
side effects 425
simulation 597
SIN (real function) 595
sort
binary tree 567
bubble 133, 622
bucket 533
exchange 529
IfSort 92
insertion 532, 552, 566
key 549
merge 293, 530
MinSort 96, 107

recursive sorting 308, 313
SelectSort 116, 292
SQR (integer, real function) 429, 595
SQRT (real function) 595
stack 370, 523, 614
standard description
(of abstract object) 485
standard words 7
state
abstract 484, 501
concrete 484, 501
execution 164, 561
state machine 342, 351
final state 352
start state 351
transition 342, 352
stepwise refinement 65
stopping rule 188
string 142
composition (∇) 145
concatenation (&) 144
decomposition 146
empty (††) 143
literal 8, 142
substring () 145
structuring a program 581
subrange data type 373, 400-402
subscript of an array variable 513
meaning 518
SUCC (ordinal function) 426
succeeds symbol (>) 24
(see relational operators)
syntax 7, 45
literal 46
metasymbol (::= < > |) 46, 47
part 46
rule 46
tree 48, 380

tag field 628
tail of a string or list (Λ) 146, 149
target value (of pointer) 560
termination
decidability 236
recursive procedures 328
WHILE statement 221, 231, 236
testing
boundary (or special) values 558
partition 279
structural 126, 279
TEXT files 110, 448
external 413
list description 150
state 151
trace table 182
conditional 200
transpose
function, relation (r^T) 168
tree
leaf 4, 557
linked structure 548, 555, 567
syntax 48
trigonometric functions 595
TRUNC (real function) 594
truth table 102
truth values
TYPE declaration 373, 417, 483, 613
type equality 417
assignment compatible 417, 421
compatible 417
same 417

union
discriminated 628
free 629
UNPACK (array operation) 534

variable
buffer 442
simple 15
subscripted 513
variant record 627
Venn diagrams 155

WHILE statement 30
design rule 229, 404
iterative meaning 217
recursive meaning 214
verification rule 220-222, 234-236
WITH statement 465
WRITE statement 10, 19, 110, 447
meaning 176
WRITELN statement 8, 19, 447
meaning 176