To the Graduate Council:

I am submitting herewith a thesis written by Pin-Ying Chou entitled "A YACC Implementation for a PL/M Interpreter." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Kevin C. O'Kane, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Chancellor
Graduate Studies and Research

A YACC IMPLEMENTATION FOR A PL/M INTERPRETER

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Pin-Ying Chou

December, 1980

Dedication

This thesis is dedicated to

my mother, Mrs. Chia-Hun Wu Wang, and to

my husband, Chang-Pin Chou.

ACKNOWLEDGEMENTS

The author wishes to express her sincere gratitude to her major professor, Dr. Kevin C. O'Kane, for his continued encouragement, advice, patience, and understanding throughout the course of investigation. Appreciation is also expressed to committee members, Dr. Charles E. Hughes and Dr. David W. Straight, for their valuable suggestions and for review of this thesis. The author wishes to express special gratitude to Dr. Charles E. Hughes for his kind guidance and help during this work.

Appreciation is also expressed to Dr. John T. Hemmeter, director of Institutional Research, for his understanding, encouragement, and assistance. The author also wishes to extend her gratitude to Dr. Suzanne W. Larsen and Mrs. Carolyn S. Thrift for their help in editing this thesis.

Final thanks are due her husband, Chang-Pin, for his patience and encouragement, and especially for the many ways in which he has contributed to the preparation of this thesis.

ABSTRACT

PINYIM, a YACC implementation of a PL/M interpreter, is presented in this thesis. The source code is written in C and runs under UNIX.

PINYIM includes five phases: lexical analyzer, parser, intermediate code generator, error handler, and execution phase. The language defined in PINYIM is a context-free language. Upon giving the grammar rules YACC produces a bottom-up LALR(1) parser. Quadruple codes are used as intermediate codes in PINYIM. These quadruple codes are executed directly if there is no error in the input stream. However, if any error is detected, the error handler will generate a diagnostic message at the user's terminal.

The language defined in PINYIM is a special dialect of PL/M. Therefore, some features of PL/M are not included and certain features are modified.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The interpreter described here takes as input a source program and produces an equivalent sequence of intermediate code which is executed directly. This process includes five phases: lexical analyzer, parser, intermediate code generator, error handler, and execution phase. The lexical analyzer, intermediate code generator, execution phase, and error handler are not very hard to implement. However, it is difficult to construct an efficient LR parser manually for a large programming-language grammar. Usually, a programming language grammar with 50 to 100 terminals and 100 grammar rules may have an LALR parsing table with several hundred states (1). Fortunately, many parser generators are available, such as YACC. YACC can build an efficient LALR bottom-up parser from a context-free language grammar.

PINYIM is a YACC implementation of a PL/M (3) interpreter. The source code is written in C (4,5) and runs under UNIX (6). The language defined by PINYIM is a context-free language and it is a subset of PL/M. Therefore, some features that may be used in PL/M are not included. The similarities and differences between these two languages are described in Chapters 5 and 6.

The grammar rules of the PINYIM language are the input for YACC. YACC then builds LALR(1) states and several large tables and writes them into a file called "y.tab.c." The parser calls the lexical analyzer to pick up the tokens from the input stream. An integer value which represents the token type is the output of the lexical analyzer. The parser groups tokens together into a syntactic structure. A semantic action code may be invoked as each structure is recognized. The intermediate quadruple code may also be generated at this time. Eventually, either an error is detected, in which case the parser returns the value 1, or an integer value 0 is returned, in which case the lexical analyzer returns the endmarker which is accepted by the parser. Under the latter case, a routine called "yyaccept" is activated and these generated quadruple codes are executed. If any error is detected during compilation, the error handler will generate a diagnostic error message at the user's terminal.

PINYIM is invoked by giving the command

```
%pinyim    pfile
```

PINYIM will then read the input stream from the "pfile." After compilation of the source program, a message is printed on the terminal which reports whether the compilation has been successfully completed. Any input or output statement in the source program is tied to the terminal.

This thesis is divided into eight chapters. In Chapter 2 an overview of the background is provided and in Chapter 3 the data structure of PINYIM are given. The descriptions of the five phases of PINYIM may be found in Chapter 4. From Chapter 5 through Chapter 7 the PINYIM language is described and some sample programs are shown. Chapter 8 contains concluding remarks. Finally, the appendices include the computer programs and grammar rules which support the program.

CHAPTER 2

OVERVIEW OF THE BACKGROUND

Introduction to YACC

YACC (Yet Another Compiler-Compiler) is written in C and runs under UNIX (2). The YACC user prepares a specification of the language to be recognized. This includes grammar rules which describe the input grammar structures, the routines which are to be invoked when these structures are recognized, and a lexical analyzer routine. YACC then produces a proceudre to do the processing. This procedure, a parser, calls the user-supplied lexical analyzer to pick up the basic tokens from the input stream. The class of grammars accepted is a very general one, called LALR(1) grammars with disambiguating rules.

The PL/M Language

PL/M is a subset of PL/I. It is a high-level language designed especially for system and applications programming for the Intel 8080 microcomputers. It was originally designed by the Intel Corporation in 1973 (3,7).

PL/M is a "block structure" language. A special feature in a PL/M program is that it is composed of blocks. Every "DO block" is a block, and every procedure declaration is a block, also. A PL/M program is made of a module which is a

particular kind of DO block. Thus, everything in a PL/M program is part of a specific block. The nested block creates a multi-level structure of blocks in a typical PL/M program.

The scope of labels and the restrictions on GOTOs are quite different from most other high-level languages. The details will be discussed in Chapter 6. The INPUT and OUTPUT statements in PL/M are two built-in procedures. They must specify one of the 256 INPUT/OUTPUT ports of the 8080 CPU.

The C Language

C is a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators (4,5). It was originally designed for and implemented on the UNIX operating system on the DEC PDP-11. The UNIX operating system, the C compiler, and essentially all UNIX application programs are written in C. C is not tied to any particualar hardware or system, however, and it is easy to write programs that will run without change on any machine that supports C. It is relatively a "low level" language which deals with the same sort of objects that most computers do such as characters, numbers, and address. C, itself, provides no input-output facilities and no wired-in file access methods. All of these must be provided by explicity-called functions.

CHAPTER 3

DATA STRUCTURE OF PINYIM

Source File Structure of PINYIM

The PINYIM source code is written in C and stored in the following files.

1. m0.c : Contains the external definitions of variables in PINYIM.
2. m1.c : Contains the external declarations of variables in PINYIM.
3. m2.c : Contains the grammar specifications of PINYIM language with the code that is invokded when the structure is recognized.
4. m3.c : Contains the parser driver routine and error handler routine.
5. m4.c
6. m5.c } : Comprises the lexical analyzer and some routines which support
7. m6.c the semantic action.
8. m7.c : Contains a routine called "yyaccept" which executes the intermediate codes when the parser successfully parses the input stream.
9. y.tab.c: Contains an output file from YACC by giving the grammar rules from file "m2.c." The parsing tables generated by YACC are also given in this file.

Implementation of PINYIM

PINYIM can be installed by running a shell procedure called "install." This shell procedure is given in Figure 1. The shell procedure first runs YACC; then compiles those files which are described in the previous section. It then makes a new file called PINYIM. Then PINYIM is ready to be invoked by a simple command

```
%pinyim     pfile
```

where "pfile" is the input source program.

Grammar Rules of PINYIM

A context-free language is used for the grammar rules of PINYIM. The grammar specifications of PINYIM are given in the source file, m2.c. Fifty-three terminals, 73 non-terminals, and 151 grammar rules are used in this grammar set. The basic syntax of each grammar rule is as follows:

```
A:    b  C  D  = {
                    action code
                };
```

The left-hand side of each grammar rule must be a non-terminal. The action code is written in C and is activated only when the structure is recognized. The left-hand side of the first grammar rule is called the start symbol which has special importance. When YACC scans the grammar rules, it defines $END as the endmarker. It also augments the

```
%YACC  M2.C
%CC  M0.C  M3.C  Y.TAB.C  M4.C  M5.C  M6.C  M7.C
%MV  A.OUT  PINYIM
```

Figure 1. A Shell Procedure "install"

grammar and defines production 0 automatically as

$accept ---> starting symbol, $END

In effect, the parser is designed to recognize the start symbol. It generally represents the largest, most general structure described by the grammar rules.

Grammar rules of PINYIM are based on PL/M grammar. TO make an easy process and avoid ambiguous usage, several grammar rules are modified and deleted. For example,

SUB1: PROCEDURE (I, J) ;

is a valid procedure declaration in PL/M, but in PINYIM the syntax of a procedure declaration is as follows:

PROCEDURE SUB1 (I, J) ;

The reason of this modification is to make it easier to distinguish a procedure name from a variable name. The modifications are discussed in detail in Chapters 5 and 6. The grammar rules of PINYIM are given in the Appendix C.

Symbol Table of PINYIM

The symbol table is used to collect information about the names appearing in the source program. The symbol table in PINYIM generally includes the following information:

1. a pointer which points to the position of the first character of the identifier;
2. attributes of the name;
3. upper limit of the array;

4. a pointer which points to the position of memory to be allocated for the identifier; and
5. flag which represents indirect or direct addressing.

This information is entered into the symbol table at various times. The string of characters denoting the identifier is entered during lexical analysis. Other information is entered in response to the declaration. The symbol table space is not reused in PINYIM. Therefore, EXTERNAL and PUBLIC attribute are not included in PINYIM.

During compilation the symbol table is searched everytime an identifier is encountered. A binary search is used as the table lookup method. Associated to each record there are two linkfields: LEFT and RIGHT. These two fields are used to link the records into a binary tree structure. Although a binary search is not the best method of approach, it is still better than linear search (8).

Memory Storage Allocation

Static storage allocation is used in PINYIM. The size and the memory location of every data item are determined at the compile-time (9). Based variables and variables with the "AT" attribute are not allocated memory storage. Consider the following declarations.

```
DECLARE  A BYTE,  B (5)  BYTE;
```

```
DECLARE  PTR  ADDRESS,  VALUE  BASED
         PTR  BYTE;
DECLARE  C  BYTE  AT (.B+2);
```

The layout of memory storage is shown in Figure 2. It shows that the variable C and the subscripted variable B(2) share one byte of storage. The based variable VALUE is not allocated storage at the compile-time. At different times during the program run it may actually be in different places in memory, since its base PTR may be changed by the program.

The memory buffer in PINYIM has an arbitrary limit of 100 integers (an integer is a two-byte element in C). The first 20 memory storages are assigned to the constant table; the next 10 memory locations are assigned to the temporaries. All the numeric constants which appear in a source program are stored in this constant table. The temporaries are used to store the result of operations temporarily. Consider the following assignment.

```
X = A * B + C * D  E * F ;
```

Figure 3 shows the sequence of quadruple statements that would be generated by the semantic action routine. Note that the temporaries may be used more than once.

Block-Structure Declaration

A PINYIM program can contain several declarations for the same identifier, and the declarations may appear at

```
DECLARE  A  BYTE, B[5] BYTE ;
DECLARE  PTR  ADDRESS, VALUE BASED PTR BYTE ;
DECLARE  C  BYTE  AT ( .B + 2 ) ;
```

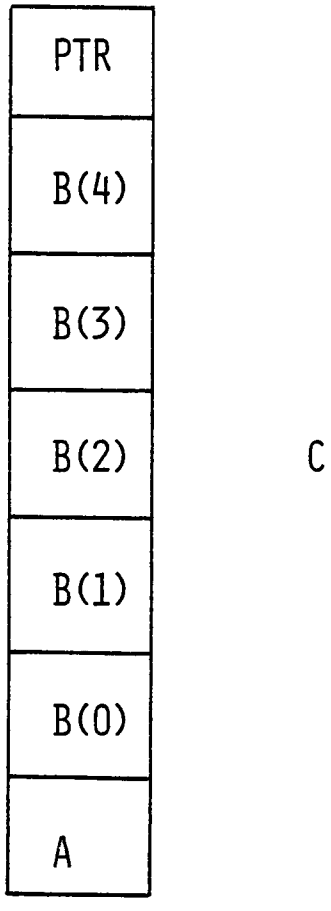

Figure 2. The Layout of Memory Storage

```
X = A * B + C * D - E * F ;
```

OPR	ARG1	ARG 2	RESULT
*	A	B	T1
*	C	D	T2
*	E	F	T3
+	T1	T2	T1
-	T1	T3	T1
=	T1		X

Figure 3. An Example of a Sequence of Quadruple Codes

the beginning of any DO block. Therefore, a special feature called block-structure declaration is used in PINYIM (10). When the parser encounters a declaration, the semantic action routine creates a table entry corresponding to that declaration and places the information contained in the declaration into the table entry. This table entry is then pushed on top of the stack for that identifier. When the DO block ends, the table entry is popped off the stack. Each table entry on a stack contains a pointer to the entry below on the stack. As an example, consider the program of Figure 4. At the time the semantic action routine is processing the assignment statement in the program of Figure 4, the symbol table structure, the array, and pushdown stacks have the form shown in Figure 5. To keep track of which variables are needed to update the stack involves making a linked list of all the table entries associated with a particular block.

Dimensional Limits in PINYIM

The limits in PINYIM are as follows:

1. maximum number of procedures is 10;
2. maximum number of identifiers is 50;
3. maximum number of DO blocks is 25;
4. maximum number of declarations is 50;
5. maximum number of constants is 20;
6. maximum number of arithmetic operations in a single statement is 10;

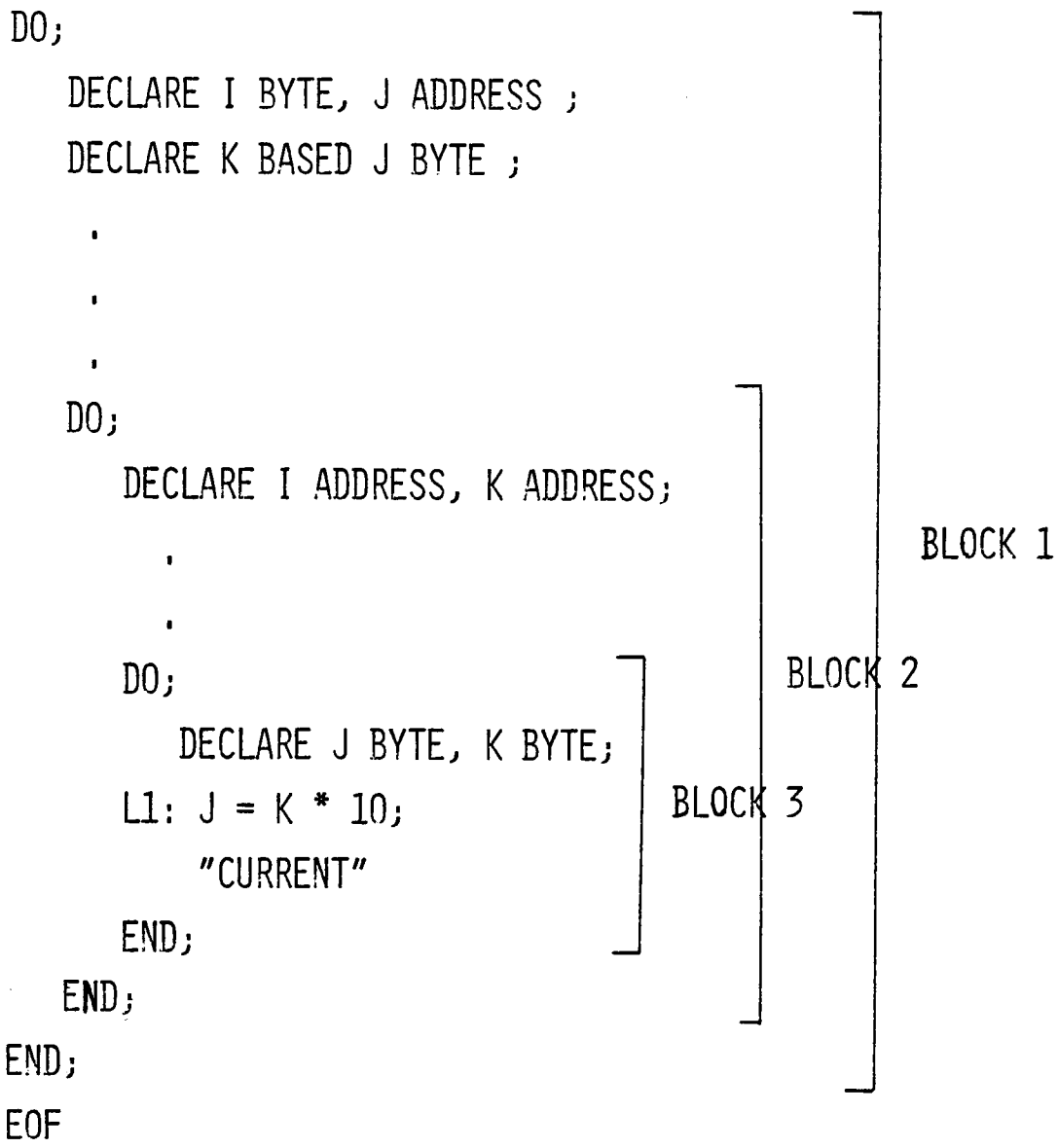

Figure 4. An Example Program

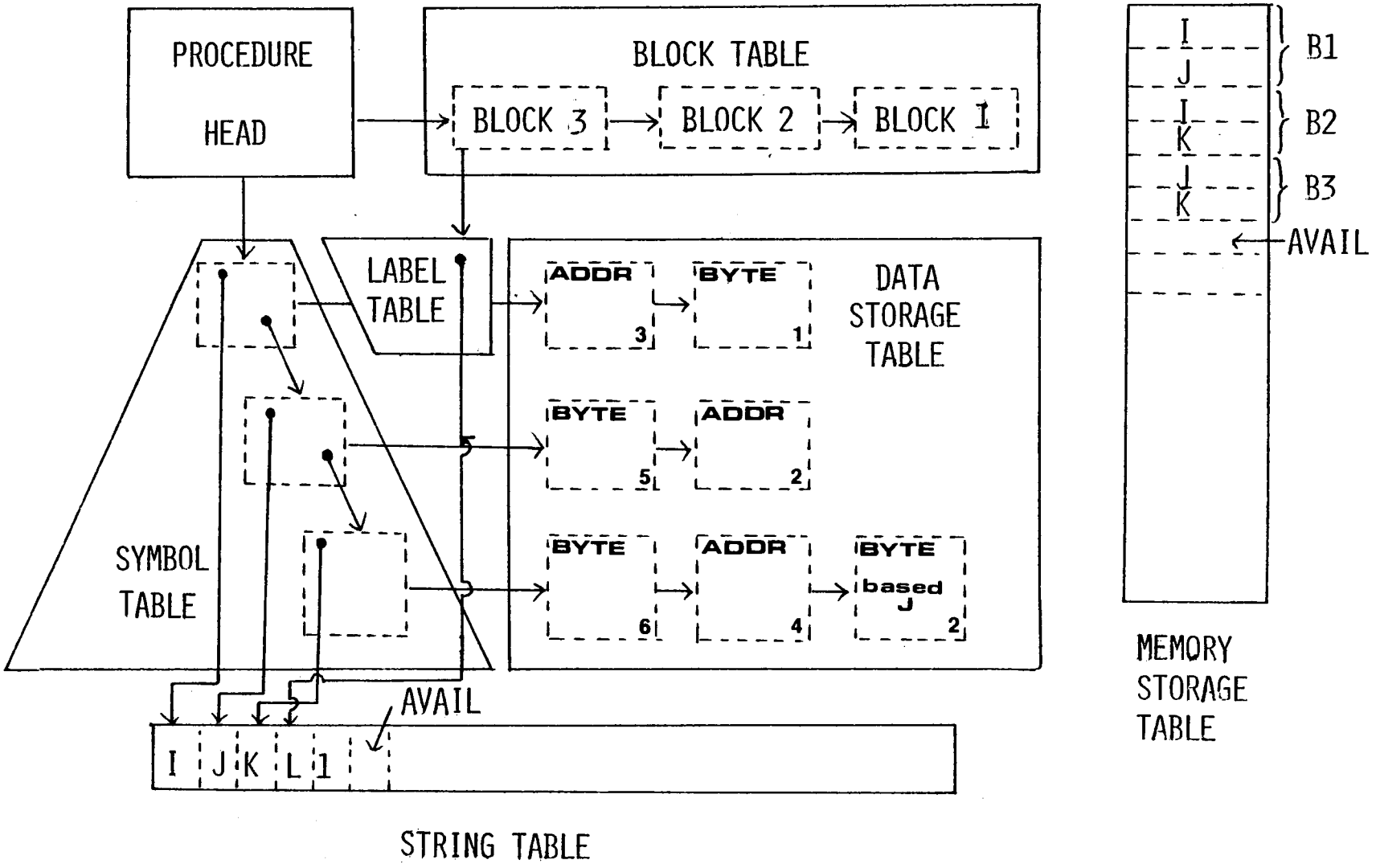

Figure 5. Symbol Table Structure for PINYIM

7. maximum number of label statements and GOTO statements is 25;
8. maximum number of units of the parsing stack is 150;
9. maximum number of units of memoey storage is 100;
10. maximum number of quadruple codes is 150.

CHAPTER 4

FIVE PHASES OF PINYIM

Lexical Analyzer

The lexical analyzer separates characters of the input stream into tokens. Fifty-three tokens can be recognized by the lexical analyzer. The lexical analyzer is an integer function called "yylex." The function returns an integer value which represents the type of the token. The type number of each token is chosen by YACC. The default type number for a literal character is the numerical value of the character. Other token names are assigned type numbers starting at 257. The type number is defined by using the "#define" mechanism in the file "y.tab.c."

There are 55 ASCII characters which could be recognized by the lexical analyzer. The list of these 55 characters is given in Chapter 5. The lexical analyzer will treat any unrecognized character as a space and report an error message to the error handler.

Parser

Upon providing the grammar rules YACC produces a bottom-up LALR(1) parser. The LALR(1) praser consists of two parts: the driver routine and the parsing table. Usually, a parsing table is composed of an ACTION table

and a GOTO table (1). The parsing table is described in detail in the remainder of this section.

The tables "yyact", "yypact", "yygo", "yypgo", "yyr1", "yyr2", "yysterm", and "yysnterm" contain the parsing table and are included in the file "y.tab.c." The "yyact" table actually is the ACTION table segment of the parsing table and the "yygo" table is the parsing GOTO table.

In the "yyact" table the following kind of actions to be taken while parsing is described:

1. ERROR: An error occurs in the current state if an improper token is found on the input stream.
2. TEST: Tests for an appropriate token on the input stream in order to shift, reduce, or accept.
3. SHIFT: Shifts to next specified state and stacks input token from the input stream.
4. REDUCE: Reduces by the appropriate grammar rule, then goes to the state indicated by the GOTO table.
5. ACCEPT: Accepts the input stream.

The actions are coded in the "yyact" table by the following scheme. To denote an error action a value of 0 is expressed in the entry of the "yyact" table.

For a test action the base code is 4096, and added to this base code will be the type number of the token invoked

for the test action. The base code for a shift action is 8192. Added to this base code will be the next-state state number. For the next case, a reduce action, the base code is 12288, and added to this base code will be the grammar rule number. Finally, for the accept condition, the base code is 16384.

The actions in the "yyact" table are accessed by pointers in the "yypact" table. The index position of a pointer is the number of a state. For example, if the parser encounters:

1. the error condition in state i;

   The position in "yypact" contains a pointer to a position in "yyact" (for state i) which contains a value of zero.

2. a test condition;

   Given the case in state 1 where, if the next input token is 'a' (type number is 257 for example), shift to state 3. Then the type number 257 is added to the test action base code to get an entry of 4353. The state number 3 is added to the shift action base code of 8192 to get a subsequent entry of 8195. Thus state 1 contains a pair of values (4353,8195). There could be many such pairs for the state. A special case for the test action is the accept condition which can only

occur in state 1. Then the entry in "yyact" for state 1 would contain a pair of values (4096,16384).

3. For a reduce action the grammar rule number is added to 12288 and, as an example for grammar rule 17, the action table entry would be 12305. The GOTO function is described with the GOTO table description.

The table "yysterm" contains only token names with type numbers greater than 256 and are numbered in the declaration section (The ASCII characters have type number less than 256). The nonterminal symbols are contained in table "yysnterm" and are numbered in the order of their appearance in the grammar rules.

"yyrl" is a table of pointers to the "yysnterm" table for the left-hand side of each grammar rule where the index value of a pointer is the grammar rule number.

The table "yyr2" contains the count of the number of right-hand side grammar symbols for each grammar rule where the index value of the count is the grammar rule number. Thus "yyrl" and "yyr2" contain exactly one entry for each grammar rule.

The table "yypgo" contains pointers to the "yygo" table. There is a one to one correspondence for each entry in "yypgo" and each nonterminal in "yysnterm" because the index of each pointer in "yypgo" is a nonterminal number.

The table "yygo" is the GOTO table of the parsing table. The "yygo" is only referenced by a REDUCE action. The "yygo" table contains pairs of values. The first number of each pair can be -1(default value); otherwise it is the current state number. The second number of the pair indicates the state to go-to after a reduction has been performed.

Interpreter Code Generator

Quadruple codes are used as intermediate codes in PINYIM. A record structure with four fields which are named OP, ARG1, ARG2, and RESULT is the typical representation of a quadruple code (1). OP contains the internal operations code. There are 23 different operation codes used in PINYIM. The list of these 23 operation codes is given in Appendix A. The quadruple code is generated when some syntactic structure is recognized. Generally, only executable statements in the source program will be translated as quadruple codes. In other words, no quadruple code will be generated for any declaration statement. Figure 3 shows a sequence of quadruple statements generated for an assignment statement.

Error Handler

The function of error handler is the detection and reporting of errors in the source program (9). Errors can be encountered in all of the phases of PINYIM. For example,

1. The lexical analyzer may be unable to proceed because the next token in the source program is misspelled.
2. The parser may be unable to infer a structure from its input because of a syntactic error such as a missing keyword.
3. The intermediate code generator may detect an operator whose operands have imcompatible types.
4. While binding the attributes to the data object, the routine may discover data declared with contradictory attributes.

Error handling is an extremely difficult area and many of the problems are semantic ones. It is generally not acceptable to stop all processing when an error is found. Thus YACC provides a simple and general strategy for the parser. The token name, "error", is reserved for error handling (2). This name can be used in grammar rules. It suggests places where errors are expected and recovery might take place. The parser attemps to find the last time in the input when the special token, "error", is permitted. To prevent a cascade of error message, the parser assumes that, after detecting an error, it remains in error state until three tokens are successfully read and shifted. If an error is detected when the parser is

already in error state, no error message is given, and the input token is quietly deleted.

Execution Phase

Those generated quadruple codes are executed directly if there is no error in the input source program. Appendix A presents a listing of operations that may be performed during the execution. The illegal GOTO statement and division by zero are detected at this time. All the variables are automatically initialized with zero.

CHAPTER 5

BASIC CONSTITUENTS OF A PINYIM PROGRAM

The language defined in PINYIM is a special dialect of PL/M. Therefore, some features of PL/M have been deleted or modified. In the following two chapters the PINYIM fundamentals are given with the assumption that the manual of PL/M has been read.

PINYIM Character Set

There are 55 acceptable characters in PINYIM. These are the alphanumerics,

ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789

special characters,

< = + - * / [] ( ) ' , . : ; >

blank characters,

space line-feed

and the dollar sign $

PINYIM will treat any character that is not in the above set as a space and produce an error message. Upper and lower case letters are not distinguished from each other, except in a string constant, in PL/M. However, only upper case letters are valid in PINYIM. Special characters and combinations of special characters have the same definition in PL/M and PINYIM.

Identifiers and Reserved Words

Identifiers are used to name variables, procedures, and statement labels. A PL/M identifier may be up to 31 characters in length and embedded dollar signs are totally ignored by the compiler. The maximum length of an identifier in PINYIM is 10 characters, and dollar signs are extremely significant. For example, INPUT$CNT and INPUTCNT are not interchangeable. A list of PINYIM reserved words is given in Appendix B. Many reserved words used in PL/M, such as DATA, are not reserved words in PINYIM.

Tokens, Separators, Comments and the Use of Blanks

There is no difference between PL/M and PINYIM concerning separators, comments, and the use of blanks. For the most part it is obvious where one token ends and the next one begins, but in some cases identifiers, reserved words, and numeric constants must be separated from each other, and a blank must be placed between them as a separator. Blanks may also be inserted freely around any token without changing the meaning of the program.

Explanatory comments are used to improve readability and provide program documentation. A sequence of characters that begin with /* and end with */ is a valid comment in PL/M and PINYIM.

Data Types

Data processed by both a PINYIM or PL/M program must be either a BYTE or an ADDRESS quantity. A BYTE variable consists of an eight-bit, unsigned integer value which is between 0 and 255. An ADDRESS variable in PL/M can hold an unsigned sixteen-bit integer value between 0 and 65535. However, in a PINYIM program, an ADDRESS variable consists of two bytes and can hold a signed integer between -32767 and 32767, inclusive.

Constants

A constant is a value known at compile time which does not change during execution of the program. A constant is either a number or a character string.

Only a decimal number is recognized as a numeric constant by PINYIM, and the dollar sign can not be inserted in a numeric constant.

Character strings are denoted by printable ASCII characters enclosed within apostrophes. Each character has an ASCII binary-coded representation. Only the first two characters of a character string are significant in both PL/M and PINYIM. In other words only the first two characters may be used as a character string constant. A one-character string is treated as a BYTE numeric constant. If it is a two-character string, it is treated as an ADDRESS numeric constant.

Variable Reference

A variable may be a simple variable or an array. Before a reference can be made to it, each variable used in a PINYIM program must be declared either as BYTE or ADDRESS in a DECLARE statement. The indirect reference, BASED variable, is a variable which is pointed to by another variable and may also be used in PINYIM. It is not allocated memory storage by the compiler. In PL/M a procedure may be used as a variable reference. However, it is not accepted in PINYIM.

Expressions

Any constant, variable reference, and location reference, may appear as operands in expressions. A location reference is formed by using the dot operator, for example, .COUNT. The value of this location reference is the address of the variable at run time.

The five arithmetic operators specified in PINYIM are as follows:

+ addition
- subtraction
* multiplication
/ division
MOD

In addition and subtraction, the result of the operation is of type ADDRESS if either operand is an ADDRESS quantity or of type BYTE if both operands are BYTE quantities. The

result of multiplication and division is always of type ADDRESS. The result of division by 0 is 0 and an error message is produced.

Relational operators are used to compare operands. They are

```
<
<=
>
>=
<>
==
```

There are 3 logical operators in PINYIM

NOT AND OR

XOR is not permitted in PINYIM.

Unlike PL/M, if the result of comparison is "true", a value of 1 is assigned; otherwise a value of 0 is assigned.

When an unparenthesized expression contains more than one operator, the precedence of the operators is governed by the following list. The operators listed below are in order from highest-to-lowest precedence operators. At the same precedence level the operations are carried out from left to right.

```
unary "-"
* / MOD
+ -
< <= <> == >= >
```

NOT

AND

OR

Assignment

The form of a PINYIM ASSIGNMENT statement is

variable = expression

or

variable = restricted expression

The first operand of a restricted expression may only be a location reference. If an operand follows the first operand, it must be a numeric constant of which + and - operators are allowed. The following examples are valid restricted expressions.

. count

. count + 2

. count - 3

When a restricted expression is used in an assignment statement, the variable on the left-hand side must be of type ADDRESS; otherwise an error message is generated.

If the type of the value of the expression is not the same as the type of the variable on the left side of an assignment statement, the value of the expression is automatically converted to match the type of the variable.

A multiple assignment statement may be used in PINYIM, for example,

variable 1, variable 2, ... = expression ;

When this is done the variables on the left are not required to be of the same type. BYTE/ADDRESS conversion is carried out as needed.

INPUT and OUTPUT Statement

The INPUT and OUTPUT statements in PINYIM and PL/M are totally different from each other. Since PL/M is specially designed for INTEL 8080, the I/O statement is related to the hardware of that machine. Either an INPUT or an OUTPUT statement must specify one of the 256 input or output port numbers. However, the I/O statement designed in PINYIM is very simple and depends on the C language. The syntax of these statements is as follows:

variable = INPUT ;

OUTPUT = variable ;

INPUT and OUTPUT are two reserved words. Once the INPUT statement is executed a message "please input number --" is printed on the user's terminal and an unsigned integer number is expected to be input from the terminal. This value is then assigned to the variable on the left hand side of the statement. If an OUTPUT statement is performed, a message "in line n output value = " followed by an output value are shown on the terminal. BYTE/ADDRESS conversion is also applied to the INPUT/OUTPUT statement.

DECLARE Statement

The DECLARE statement is a nonexecutable statement. Each variable used in a PINYIM program must appear in a DECLARE statement before reference can be made to it by its identifier. All the DECLARE statements in a DO block must appear at the beginning of the block.

The scope of a declared object is limited to the block in which it occurs. If any sub-block nested within the block contains a declaration which declares the same identifier, then the scope defined by the outer declaration excludes the sub-block. The variable declaration may be used only in a PINYIM program. The use of a label declaration is invalid in PINYIM.

Some restrictions are made on the DECLARE statement in PINYIM. The basic syntax of a variable declaration is as follows:

```
DECLARE name [dimension specifier] type [attribute]
             [base specifier]  [initialization] ;
```

An example is

```
DECLARE C$VALUE BYTE ;
```

A base specifier has the form

```
BASED base-identifier
```

The following is an example of a declaration of a based variable.

```
DECLARE C$VALUE BASED VALUE$PTR BYTE ;
```

In a PINYIM program the dimension specifier is a numeric constant in brackets instead of in parentheses. An implicit dimension may be used in PL/M, but it is not valid in a PINYIM program.

Each PINYIM variable must be declared with a "type." There are two types: BYTE and ADDRESS. The STRUCTURE type may not be used in a PINYIM program.

The only attribute that can be used in PINYIM is the AT attribute. The AT attribute has the following form:

AT (restricted expression)

The definition of a restricted expression is given in Chapter 4. The variable declared with the AT attribute is not allocated storage by the compiler. The AT attribute can be used to make variables "equivalent", thus providing more than one way of referring to the same variable.

Initialization is used to supply values for variables at compile time.

The declaration

DECLARE VALUE [5] BYTE INITIAL (2,4,6,8,13) ;

declares the BYTE array VALUE and initializes its five elements to 2, 4, 6, 8, 13. In PINYIM there is no DATA initialization.

A separate DECLARE statement is not required for each and every declaration. Instead of writing the two

DECLARE statements

```
DECLARE A BYTE ;
DECLARE B ADDRESS ;
```

Both declarations may be written in a single DECLARE statement, like this :

```
DECLARE A BYTE, B ADDRESS ;
```

however, the factored declaration may not be accepted in PINYIM. for example,

```
DECLARE ( A , B ) BYTE ;
```

CHAPTER 6

BLOCKS AND TRANSFER STATEMENTS

This chapter includes a description of the blocks and statements that affect the sequence of executable statements in a PINYIM program. Like PL/M, the PINYIM language is a "block structured" language. Do block and PROCEDURE are two kinds of blocks used in PINYIM. The important property of all blocks is to control the scope of the objects declared in the program. The entire program is a special kind of DO block and it is called a module.

DO Blocks

Simple DO block, DO WHILE block, iterative DO block, and DO CASE block are four kinds of DO blocks. The syntax of each DO block in both languages is the same. However, except for the simple DO block, modifications exist in DO blocks implemented within the PINYIM language.

Figure 6 gives an example of a simple DO block. Each statement within a simple DO block may be an executable statement or a declaration, with the restriction that all declarations within the outer level of the DO block must appear before the first executable statement that occurs at the outer level. A simple DO block may be regarded as a single and may appear anywhere in a program that a single executable statement may appear.

```
DO;
  DECLARE CURRENT ADDRESS, COUNTER BYTE;
  DECLARE VALUE BASED CURRENT BYTE ;
  IF VALUE  >= 50
  THEN
     VALUE = VALUE MODE 10 ;
  ELSE
  DO;
     COUNTER = COUNTER + 1;
     VALUE = VALUE * 10 ;
  END;
END:
```

Figure 6. An Example of a Simple DO Block

An example of a DO WHILE block is shown in Figure 7. The expression following the keyword WHILE is evaluated first. In a PINYIM program if the result is true a value 1 is assigned, otherwise a 0 is assigned. The block is executed over and over until the expression has a value 0. However, in PL/M DO WHILE and IF statements examine only the least significant bit of the value of the expression. If the value is an odd number (least significant bit = 1), it will be considered true. If it is even (least significant bit = 0), it will be considered false.

Figure 8 illustrates an example of an iterative DO block. A special feature of an iterative DO block in PL/M is that if incrementing the index variable causes the new value to be less than the old value, control passes immediately to the statement following the END statement. However, PINYIM does not test the transition of the index variable value during the execution of an iterative DO block.

PINYIM makes two changes on the DO CASE statement. These are:

1. The null statement (empty statement) is not accepted in a PINYIM program.
2. If the run-time value of the expression in the DO CASE statement is greater then the number of cases in the DO CASE block, then no one statement within the block is executed.

A DO CASE example is given in Figure 9.

```
AMOUNT = 1 ;
DO WHILE AMOUNT>= 3 ;
     OUTPUT = AMOUNT ;
     AMOUNT = AMOUNT + 1 ;
END;
```

Figure 7. An Example of a DO WHILE Block

```
DO I = 0 TO 10 BY 2 ;
     OUTPUT = VALUE (I) ;
     DO WHILE VALUE (I) > 5;
          CALL ADD ;
     END;
END;
```

Figure 8. An Example of an Iterative DO Block

```
I = INPUT ;
DO CASE I - 2 ;
     OUTPUT = VALUE(I) ;        /* CASE 0 */
     DO;                        /* BEGIN CASE 1 */
        X = X + 1;
        Y = Y + 1;
     END;                       /* END CASE 1 */
     CALL DUMP (.VALUE);        /* CASE 2 */
END;                            /* END DO CASE BLOCK */
```

Figure 9. An Example of a DO CASE Block

A DO WHILE block, an iterative DO block, and a DO CASE block can be considered a single statement just like a simple DO block. However, unlike a simple DO block, the other block statements may not contain declarations at the outermost level.

Syntax of a PROCEDURE Declaration

A procedure declaration consists of three parts: a procedure statement, a procedure body, and an END statement. Figure 10 includes a simple example of a procedure declaration. This procedure sums the first N+1 elements of the BYTE array pointed to by PTR.

Some modifications have been made. The procedure statement begins with a reserved word PROCEDURE in the PINYIM language. The reason for this modification is to assist in distinguishing a procedure name from a variable name. A function procedure may not be accepted in a PINYIM program.

The procedure body is always a simple DO block. Upon giving a simple DO block, the procedure block is an explicit DO block, and the scope of labels and GOTO statements are processed easily. However, in PL/M any statement may be included in a procedure body.

If the procedure has no formal parameters, the parameter list is omitted from the procedure statement. When a procedure has formal parameter, each formal parameter must be declared as a nonbased variable in a DECLARE statement

```
PROCEDURE SUM$ARRAY ( PTR,N );
DO;
  DECLARE PTR ADDRESS, N BYTE;
  DECLARE ARRAY[5] BASED PTR BYTE;
  DECLARE I BYTE;
  SUM = 0;
  DO I = 0 TO N;
     SUM = SUM + ARRAY(I) ;
  END;
  RETURN;
END;
END SUM$ARRAY;
```

Figure 10. An Example of a PROCEDURE Declaration

preceding the first executable statement in the procedure body. The formal parameters and local variables are both allocated memory storage during the compilation. However, the storages are not released when the procedure is exited.

The execution of a procedure is terminated by one of two ways:

1. By execution of a RETURN statement within the procedure body.
2. By executing a GOTO to a statement outside the procedure body. The target of the GOTO must be at the outer level of the main program.

In PL/M a procedure may also be terminated by reaching the END statement.

PROCEDURE Calls

There is only one possible way to make a procedure call in PINYIM. It is called by means of a CALL statement which is the form

```
CALL   name  (parameter list) ;
                or
CALL   name ;
```

An example is the following:

```
CALL   SUM$ARRAY  (.A , 4) ;
```

The procedure SUM$ARRAY shown in Figure 10 may be invoked by this CALL statement. At the time of the call each actual parameter is evaluated and its resulting value is assigned

to the corresponding formal parameter. As in the above example, the memory location of variable A is passed to the formal parameter PTR, and the constant four is assigned to the variable N. The type of formal and actual parameters must be matched.

Parameter-Passing Mechanism

When passing a parameter, call-by-value is used both in PINYIM and PL/M. Without using the BASED variable, a parameter may not be used to return a value to the calling program and an array may not be processed in the procedure by just passing the name of the array. Figure 11 shows a simple procedure that dumps any given array. Its parameter has been named ARRAY$PTR. This parameter is immediately declared as being of type ADDRESS and is used in declaring ARRAY as being a based array having 10 elements. This last declaration does not cause the allocation of any storage. In different invocations of this procedure, many different arrays at varying locations can be dumped.

IF Statement

The IF statement provides conditional execution of statements. An example is given in Figure 12. Unlike the DO statement, PINYIM implements the IF statement exactly as does PL/M. IF statements may be nested within one another, and the ELSE part is optional. The power of the IF statement is enhanced by using DO blocks in the THEN and ELSE

```
PROCEDURE DUMP (ARRAY$PTR);
DO;
  DECLARE ARRAY$PTR ADDRESS;
  DECLARE ARRAY[10] BASED ARRAY$PTR BYTE;
  DECLARE J BYTE;
  DO J = 0 TO 10;
     OUTPUT = ARRAY(J) ;
  END;
END;
END DUMP;
```

Figure 11. An Example of a BASED Variable in a Procedure

```
IF A <B  THEN
DO;
   LARGER = A ;
   A$COUNT = A$COUNT + 1;
END;
ELSE
DO;
   LARGER = B ;
   B$COUNT = B$COUNT + 1;
END;
```

Figure 12. An Example of an IF Statement

parts since a DO block is allowed wherever a single statement is allowed.

GOTO Statement

Before the rules on the GOTO statement are discussed, two terminologies used in PL/M must be mentioned here. These are inclusive and exclusive extent.

The inclusive extent of a block is everything from the DO or PROCEDURE statement that begins the block to the END statement that terminates it. The DO or PROCEDURE statement and the END statement are considered to be part of the inclusive extent of a block. Any label on a DO statement is not part of the inclusive extent. Figure 13 presents the inclusive extent of a block. Everything inside the solid line constitutes the inclusive extent of block M. Everything inside the dashed line constitutes the inclusive extent of block SORT. Everything inside the dotted line constitutes the inclusive extent of block FIND.

The exclusive extent of a block is the inclusive extent of that block minus the inclusive extent of all nested blocks. In Figure 14 the shaded area is the exclusive extent of block SORT.

The restrictions on the GOTO statement defined in PL/M are exactly followed in the PINYIM language. These are:

INCLUSIVE EXTENT OF A BLOCK

```
M: DO;      /* BEGINNING OF MODULE */

      DECLARE TABLE [10] BYTE ;

      DECLARE TEMP BYTE;

      DECLARE I BYTE, J BYTE;

SORT: DO J = 1 TO 9 ;
        TEMP = TABLE (J) ;
        I = J;

FIND:   DO WHILE I > 0 AND TABLE(I-1) > TEMP;
          TABLE(I) = TABLE(I-1);
          I = I-1 ;
        END FIND;

        TABLE(I) = TEMP ;
      END SORT;

END M;      /* END OF MODULE  */
```

Figure 13. An Example of an Inclusive Extent of a Block

EXCLUSIVE EXTENT OF A BLOCK

```
M: DO;     /* BEGINNING OF MODULE */

      DECLARE TABLE [10] BYTE ;

      DECLARE TEMP BYTE;

      DECLARE I BYTE, J BYTE;

SORT: DO J = 1 TO 9 ;
         TEMP = TABLE (J) ;
         I = J;

FIND:    DO WHILE I > 0 AND TABLE(I-1) > TEMP;
            TABLE(I) = TABLE(I-1);
            I = I - 1 ;
         END FIND;

         TABLE(I) = TEMP ;
      END SORT;

END M;     /* END OF MODULE  */
```

Figure 14. An Example of an Exclusive Extent of a Block

1. from a statement in the exclusive extent of a block to a statement in the exclusive extent of the same block.
2. from an inner block to a statement in the exclusive extent of an enclosing block. However, if the inner block is a procedure block, the transfer may only be to a labeled statement in the exclusive extent of the main program.

CALL and RETURN Statement

The CALL statement is used to activate a procedure. Upon giving a CALL statement, control passes to the procedure. Both PL/M and PINYIM use call-by-value as the parameter-passing mechanism. The details are mentioned earlier in this chapter.

The RETURN statement is used within a procedure body to cause a return from the procedure to the point from which it was called. Since function procedures are not accepted in PINYIM, there is no possible return of a value associated with executing a RETURN statement.

CHAPTER 7

SAMPLE PROGRAMS

At this point all of the constructions available in PINYIM except a complete program have been discussed. Therefore, three sample programs are illustrated this chapter.

Figure 15 presents the first example which implements a simple sort algorithm. The source program is stored in the file "t2." The effect of the algorithm is to arrange the I-th element through J-th element of the array B in order according to the values assigned to these elements. The variable B is declared as a BYTE array having six elements. The first element of the array B is B(0), and the last element of the array B is B(5). The first five elements of the array B are initialized by the first DECLARE statement in the main program. However, B(5) is automatically initialized with zero. The values of variables I and J are assigned by two INPUT statements. These two values are then passed to the procedure SORT.

The result of the program is shown in Figure 16. After compilation of the source program, a message "no compile-time error" is printed on the user's terminal. Two input values for variable I and J are typed on the terminal after the message "please input number --" is

```
/*****************************************
 ***   SAMPLE PROGRAM NO. 1            ***
 *****************************************/
SORT$PRG:
DO;
DECLARE B[6] BYTE INITIAL ( 8,5,10,2,6);

     /*  THIS PROCEDURE IS USED TO SORT THE ARRAY B */

   PROCEDURE SORT (I,J);
   DO;  DECLARE I BYTE, J BYTE;
        DECLARE H BYTE, K BYTE;
        DECLARE T BYTE;
        DO H = I TO J;
        DO K = H +1 TO J ;
           IF B(H) > B(K)
           THEN DO;
                T = B(H);
                B(H) = B(K) ;
                B(K) = T ;
           END;
         END;
         END;
         RETURN;
   END;
   END SORT;

/*  THIS IS THE MAIN PROGRAM  */

  DECLARE I BYTE, J BYTE;

  /*  INPUT TWO VALUES FOR I AND J */

   I = INPUT;
   J = INPUT;

  /*  CALL SORT ROUTINE */

   CALL SORT ( I,J);

/*  PRINT OUT THE SORTED ARRAY */

DO I = 0 TO 5 ;
   OUTPUT = B(I);
END;
END;
EOF
```

Figure 15. SORT Program

```
% pinyin t2
  no compile-time error
 please input number--0
 please input number--5
 in line 44 output value=0
 in line 44 output value=2
 in line 44 output value=5
 in line 44 output value=6
 in line 44 output value=8
 in line 44 output value=10
%
```

Figure 16. The Output of SORT Program

seen. By executing the OUTPUT statement within the iterative DO loop, the sorted elements of the array B are printed.

Figure 17 shows the second sample program which calls the procedures MEAN and DUMP. The procedure MEAN computes the mean value of an array, and the procedure DUMP is used to dump the contents of any giving array. The source program is stored in the file "t6." In the main program, the procedures MEAN and DUMP are activated by the CALL statements. The actual parameter of each CALL statement is a location reference. The parameter passing mechanism is call-by-value in PINYIM. Therefore, in order to process an array in a procedure, the address of the array must be passed as an actual parameter and that value used as the base of a based variable.

Within these two procedures the formal parameter ARRAY$PTR is immediately declared as being of type ADDRESS and is used in declaring ARRAY as being a based array having only one element. Since no storage is allocated to the array in the procedure, it makes no difference how many elements are specified in the declaration. However, some size must be specified so that PINYIM will know that the variable ARRAY is dealing with an array and will therefore permit subscripts. It is immaterial what dimensioning information is given. Whenever the procedure refers to

```
/*************************************************************
 ***                  SAMPLE PROGRAM NO. 2                 ***
 *************************************************************/
MAIN:
DO;
    /*  COMPUTES THE MEAN VALUE OF ANY GIVING ARRAY  */

   PROCEDURE  MEAN ( ARRAY$PTR, J);
   DO;
     DECLARE ARRAY$PTR ADDRESS, ARRAY[1] BASED ARRAY$PTR BYTE;
     DECLARE J BYTE;
     DECLARE I BYTE, TEMP BYTE;
     DECLARE MEAN$VAL BYTE;
     TEMP = 0;
     DO I = 0 TO J;
        TEMP = TEMP + ARRAY(I);
     END;
     MEAN$VAL = TEMP / (J+1);
     OUTPUT = MEAN$VAL;
     RETURN;
   END;
   END MEAN;

   /*  DUMPS ANY GIVING ARRAY  */

   PROCEDURE DUMP ( ARRAY$PTR, J);
   DO;
     DECLARE ARRAY$PTR ADDRESS, ARRAY[1] BASED ARRAY$PTR BYTE;
     DECLARE J BYTE;
     DECLARE I BYTE;
     DO I = 0 TO J;
        OUTPUT = ARRAY(I);
     END;
     RETURN;
  END;
  END DUMP;

 /* MAIN PROGRAM */

DECLARE TABLE[5] BYTE;
DECLARE I BYTE;
  DO I = 0 TO 4;
     TABLE(I) = INPUT;
  END;

  CALL DUMP (.TABLE, 4);
  CALL MEAN (.TABLE, 4);
  DO I = 0 TO 4;
     TABLE(I) = TABLE(I) + 5;
  END;

  CALL DUMP (.TABLE, 4);
  CALL MEAN (.TABLE, 4);
END;
EOF
Z
```

Figure 17. MEAN and DUMP Program

ARRAY, it actually processes whatever array in the main program is pointed to by ARRAY$PTR.

TABLE is declared as an array having five elements in the main program. First, the main program initializes the array TABLE by executing a DO loop which contains an INPUT statement; next, the contents of the array TABLE are dumped. Then, the mean value of the array TABLE is computed. Finally, each element of TABLE is incremented by five and a mean value is computed again. The result of this program is given in Figure 18.

Figure 19 presents a cipher program which has some syntax errors. The error messages generated by PINYIM are shown in Figure 20.

Very little time is required for executing each of the above programs. The SORT program requires 10 seconds, the MEAN and DUMP program uses 20 seconds, and the CIPHER program requires 15 seconds for compilation.

```
% pinyim t5
  no compile-time error
 please input number--20
 please input number--23
 please input number--34
 please input number--25
 please input number--26
 in line 32 output value=20
 in line 32 output value=23
 in line 32 output value=34
 in line 32 output value=25
 in line 32 output value=26
 in line 20 output value=25
 in line 32 output value=25
 in line 32 output value=28
 in line 32 output value=39
 in line 32 output value=30
 in line 32 output value=31
 in line 20 output value=30
%
```

Figure 18. The Output of MEAN and DUMP Program

```
/****************************************
 *     SAMPLE PROGRAM NO. 3             *
 ****************************************/
MAIN: DO;
     DECLARE  TABLE [5] BYTE INITIAL ( 1,2,3,4,5);
     PROCEDURE CIPHER (C) ;
     DO;
         DECLARE C ADDRESS, CHAR BASED C BYTE;
         DECLARE CIPHERTAB [5] BYTE INITIAL (11,2,9,8,5);
         DECLARE SUBSTI [5] BYTE INITIAL (153,154,155,156,157);
         DECLARE I BYTE;
         DO WHILE CHAR <> CIPHERTAB(I) AND
                   I <= 4;
               I = I + 1;
         END;
         CHAR = SUBSTI (I);
         RETURN;
      END;
      END CIPHER;


    DECLARE PTR BYTE;
    PTR = .TABLE + 4;
    CALL CIPHER (PTR) ;
    OUTPUT = TABLE(4)
   END;
  EOF
```

Figure 19. CIPHER Program

```
% pinyim t4
 compile-time error messages :
 invalid variable type, line 23, on input : PTR
 unmatched parameter type , line 24, on input: PTR
 syntax error , line 26, on input: END
 probably missing ';'
 syntax error , line 27, on input: ;
 syntax error , line 28, on input: EOF
 please correct errors and try it again !!
%
```

Figure 20. The Output of CIPHER Program

CHAPTER 8

CONCLUSION

This thesis presents an implementation of YACC. It can be concluded by showing an architecture of PINYIM phases in Figure 21. The language defined in PINYIM is a dialect of PL/M. Some features of PL/M are not included in PINYIM and certain features that are included are slightly restricted or modified.

Summary of PL/M Features Not Included in PINYIM

1. PL/M built-in functions are not included.
2. The STRUCTURE and LITERALLY data-type are not included.
3. The EXTERNAL and PUBLIC attributes are not included.
4. The DATA initialization is not included.
5. The factored declaration and label declaration are not included.
6. The implicit dimensional specifier is not included.
7. The function procedure is not included.
8. The NULL statement is not included.

Additional Restrictions Imposed by PINYIM

1. Only 55 ASCII characters may be recognized in PINYIM.
2. The length of an identifier is restricted to 10 characters.

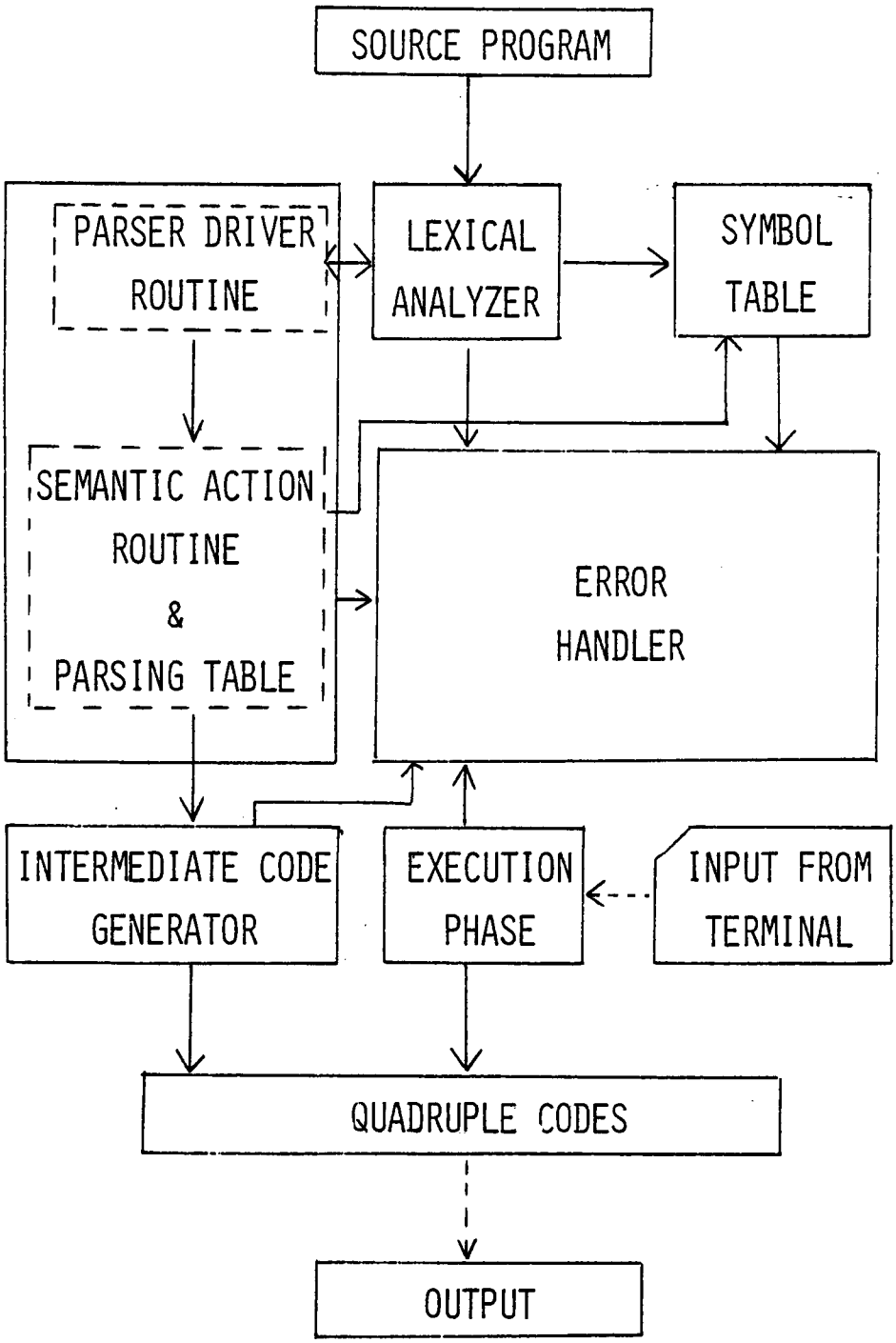

Figure 21. An Architecture of PINYIM Phases

3. There are no binary, octal, and hexadecimal numbers.
4. An ADDRESS variable may assume value from -32767 to 32767.
5. The procedure body must be a simple DO block.
6. The END statement may not be used as a statement to exit from a procedure.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. Aho Alfred V., Ullman Jeffrey D., Principles of Compiler Design, Addison-wesley Publishing Co., 1977

2. Johnson, Stephen C., YACC-Yet Another Compiler-Compiler, Bell Laboratories, Murry Hill, N. J., 1975

3. PL/M-80 Programming Manual, Document Number 98-268B, Intel Corporation, 1976

4. Ritchie, D. M., C Reference Manual, Bell Laboratories, Murray Hill, N. J., 1975

5. Kernighan, Brian W., Ritchie, Dennis M., The C Programming Language, Prentice-Hall, Inc., 1978

6. Ritchie, D. M., Thompson, K., The UNIX Time-Sharing System, Bell Laboratories, Murry Hill, N. J., 1975

7. McCracken, Daniel D., A Guide to PL/M Programming for Microcomputer Applications, Addison-Wesley publishing Co., 1978

8. Knuth, Donald E., The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley Publishing Co., 1973

9. Gries, David, Compiler Construction for Digital Computer, John Wiley & Sons, Inc., 1976

10. Lewis, P. M., Rosenkrantz, D. J., Stearns, R. E., Compiler Design Theory, Addison-Wesley Publishing Co., 1978

APPENDIXES

APPENDIX A

PINYIM INTERMEDIATE OPERATION CODES

OPERATION CODE	MEANING
1	addition
2	subtraction
3	multiplication
4	division
5	OR
6	AND
7	less than
8	greater than
9	equal to
10	less than or equal to
11	greater than or equal to
12	not equal to
13	branch
14	assign
15	test "true" or "faluse"
16	MOD
17	NOT
18	unary minus
19	control passes to a procedure
20	pass parameter
21	return
22	input
23	output

APPENDIX B

PINYIM RESERVED WORDS

IF THEN ELSE	conditional tests and alternative execution
DO END	statement grouping
DECLARE BYTE ADDRESS INITIAL BASED PROCEDURE AT	declarations
GOTO GO TO BY CASE WHILE	unconditional branching and flow control
CALL	procedure call
RETURN	procedure return
OR AND NOT	boolean operators
MOD	remainder after division
EOF	end of input file (optional)

APPENDIX C

GRAMMAR RULES OF PINYIM

```
program                 : module
                        ! module  EOF ;

module                  : module.name  simple.do.block ;

module.name             : label.definition ;

label.definition        : ID  ';'  ;

simple.do.state.head    : simple.do.statement  decpart  ;

simple.do.block         : simple.do.state.head  unit
                          ending
                        ! simple.do.state.head  ending
                        ! simple.do.statement  ending
                        ! simple.do.statement  unit
                          ending  ;

unit                    : statement
                        ! unit  statement
                        ! unit  error  ';'  ;

statement               : basic.statement
                        ! if.statement
                        ! label.definition  statement  ;

decpart                 : declaration
                        ! decpart  declaration  ;

declaration             : declaration.statement  ';'
                        ! procedure.definition  ;

basic.statement         : assignment  ';'
                        ! assignment  error  ';'
                        ! return.statement
                        ! call.statement
                        ! input.statement  ';'
                        ! input.statement  error  ';'
                        ! output.statement  ';'
                        ! output.statement  error  ';'
                        ! do.block
                        ! goto.statement  error  ';'
                        ! goto.statement  ';' ;
```

```
if.statement            : if.clause  statement
                        ! if.clause  true.part  statement ;

if.clause               : IF  expression  THEN
                        ! IF  expression  error  ;

true.part               : basic.statement  ELSE  ;

group                   : group.head  ending  ;

group.head              : DO  step.definition
                        ! DO  while.clause  ';'
                        ! DO  case.selector  ';'
                        ! group.head  statement ;

do.block                : group
                        ! simple.do.block ;

step.definition         : simple.variable  replace
                          expression iteration.control ;

iteration.control       : TO  expression  ';'
                        ! TO  expression  error  ';'
                        ! TO  expression  BY  expression
                          error  ';'
                        ! TO  expression  BY  expression ';' ;

simple.do.statement     : DO  ';' ;

while.clause            : WHILE  expression  ;

case.selector           : CASE  expression  ;

procedure.head          : procedure.name  ';'
                        ! procedure.name  parameter.list
                          ';'  ;

procedure.name.head     : PROCEDURE ;

procedure.name          : procedure.name.head  ID ;

procedure.definition    : procedure.head  decpart  unit
                          ending
                        ! procedure.head  unit  ending
                        ! procedure.head  decpart  ending
                        ! procedure.head  ending  ;

parameter.list          : parameter.head  variable  ')'
                        ! parameter.head  locator  ')'
                        ! parameter.head  NUMBER  ')' ;
```

```
parameter.head          : '('
                        ! parameter.head  locator  ','
                        ! parameter.head  NUMBER  ',' ;

ending                  : END ';'
                        ! END  ID  ';'
                        ! label.definition  ending  ;

return.statement        : RETURN  ';'  ;

call.statement          : call.head  ';'
                        ! call.head  parameter.list ';'  ;

call.head               : CALL  ID  ;

goto.statement          : goto  ID  ;

goto                    : GO  TO
                        ! GOTO  :

declaration.statement   : DECLARE  declaration.element
                        ! declaration.statement  ','
                          declaration.element ;

declaration.element     : ID  type
                        ! array.apecifier  type
                        ! declaration.element  initial.list ;

array.specifier         : array.head  NUMBER  ']' ;

array.head              : ID  '['  ;

type                    : variable.type
                        ! based.variable  basic.type  ;

variable.type1          : basic.type  ;

variable.type2          : variable.type1  AT  '('  ;

variable.type           : variable.type1
                        ! variable.type2  restr.expression
                          ')'  ;

basic.type              : BYTE
                        : ADDRESS  ;

based.variable          : BASED  ID  ;
```

```
initial.list          : initial.head  NUMBER  ')'  ;

initial.head          : INITIAL  '('  ;

assignment            : variable  replace  expression
                      ! left.part  assignment  ;

input.statement       : variable  replace  INPUT ;

output.statement      : OUTPUT  replace  variable ;

replace               : '=' ;

left.part             : variable  ',' ;

expression            : logical.expression
                      ! restr.expression ;

locator               : '.'  ID ;

restr.expression      : locator '+' NUMBER
                      ! locator '-' NUMBER
                        locator ;

logical.expression    : logical.factor
                      ! logical.expression  OR
                        logical.factor  ;

logical.factor        : logical.secondary
                      ! logical.factor  AND
                        logical.secondary  ;

logical.secondary     : logical.primary
                      ! NOT  logical.primary ;

logical.primary       : arith.expression
                      ! airth.expression relation
                        arith.expression ;

relation              : '=='
                      ! '<'
                      ! '>'
                      ! com ;

com                   : '<='
                      ! '>='
                      ! '<>'  ;
```

```
arith.expression    : term
                    ! airth.expression '+'
                      term
                    ! arith.expression '-'
                      term ;

term                : primary
                    ! term '*'  primary
                    ! term  '/'  primary
                    ! term  MOD  primary  ;

primary             : constant
                    ! variable
                    ! sub.expression
                    ! '-'  primary  ;

sub.expression      : '('  expression  ')'  ;

variable            : simple.variable
                    ! subscript.head  expression ')' ;

constant            : NUMBER
                    ! STRING ;

subscript.head      : simple.variable  '(' ;

simple.variable     : ID ;
```

APPENDIX D

PINYIM SOURCE FILES

1. m0.c (pp. 74-77): Contains external definitions of variables in PINYIM.
2. m1.c (pp. 78-81): Contains external declarations of variables in PINYIM.
3. m2.c (pp. 82-94): Contains the grammar rules of PINYIM language with the semantic action codes.
4. m3.c (pp. 95-99): Contains the parser driver routine and error handler routine.
5. m4.c (pp.100-104): Contains the lexical analyzer routine.
6. m5.c (pp.105-119) & 7. m6.c (pp.120-132): Contains some routines which support the semantic action.
8. m7.c (pp.133-138): Contains the execution phase.
9. y.tab.c (pp.139-155): Contains an output file from YACC by giving the grammar rules from file "m2.c." The parsing tables generated by YACC are also given in this file.

```
#define  MEMSIZE  100   /*  maxi number of units of memory storage     */
#define  MAXLINE  150   /*  maxi number of quadruple statements        */
#define  MAXDEPTH 150   /*  maxi number of units of the parsing stack  */
#define  TNDSIZE  50    /*  maxi number of units of the symbol table   */
#define  PNDSIZE  10    /*  maxi number of procedures                  */
#define  DNDSIZE  50    /*  maxi number of declarations                */
#define  DOBKSZ   25    /*  maxi number of DO blocks                   */
#define  DCLSIZE  50    /*  maxi number of identifiers                 */
#define  LABSTSZ  25    /*  maxi number of label and goto statements   */
#define  IDLEN    10    /*  maxi number of characters of a variable    */
#define  CONSTLEN 10    /*  maxi number of digits of a number          */
#define  TEXTLEN  100   /*  maxi number of units of the string table   */
#define  NULL     0
/*************************************************************
 ***      EXTERNAL DEFINITIONS OF VARIABLES IN PINYIN      ***
 *************************************************************/

struct   intrnode {
         int opcode;
         int opr1;
         int opr1_t;
         int opr2;
         int opr2_t;
         int result;
         int result_t; } intrtab [MAXLINE] ;

struct   lnode {
         char *lsword ;
         int  intr_ptr;
         struct lnode *bck_link;
         struct  lnode  *lnd_link; } lstab [LABSTSZ] ;

struct   dnode {
         int  dtype ;
         int  dim ;
         struct dnode *dlink ;
         int mem_ptr ;
         int mem_t;
         char *s_ptr; } dndtab[DNDSIZE] ;

struct   tnode {
         char  *tword;
         struct dnode *dnd_ptr ;
         struct tnode *left ;
         struct tnode *lf_back ;
         struct tnode *right;
         struct tnode *rg_back; } tndtab [ TNDSIZE ];

struct   dclnode {
```

```
        struct  tnode *tptr ;
        struct  dclnode *next; } dcltab [DCLSIZE] ;

struct  donode {
        char  *lword;
        struct dclnode *dcl_link ;
        struct  lnode  *lsta;
        struct  lnode  *gosta;
        struct  donode *up_lev ;
        int     dotype; /* x0000 - simple do block */
                        /* x1000 - DO case   block */
                        /* x2000 - do iterate block */
                        /* x3000 - do while  block */
        int     inx_mem;
        char    mt;
        int     intr_p;
        int     delt ;
        }  dotab [DOBKSZ] ;

struct  pnode {
        char  *pword ;
        int   ptype ;
        struct tnode *locvar;
        struct tnode *form_var;
        int     intrpnt;
        struct pnode *child ;
        struct pnode *subling ;
        struct donode *doptr ;
        struct donode *cur_do;
        struct  lnode   *lsptr;
        struct  lnode   *gsptr;
        struct pnode *ret_addr ; } pndtab [ PNDSIZE ] ;

struct  parsnode {
        int s;
        int v;
        struct tnode *tref;
        struct pnode *pref;
        int           mref;
        char         *sref;
        int           mref_t; } parstab [MAXDEPTH] ;

int     retstk[20] ;
int     *rstkptr   retstk;
int     intralc   0;
int     intrstk [20] ;
int     istkptr   0;
int     tmpstk [20];
int     tstkptr   0;
int     yymref;
int     yymrefs;
```

```
int     yym_t;
char    *yysref;
char    *yysrefs ;
int     yyline;
struct  parsnode *pars_p  parstab  ;
struct  donode *do_alc    dotab ;
struct  dclnode *dcl_alc  dcltab ;
struct  lnode   *l_alc    lstab ;
struct  tnode *yytref;
struct  tnode *yytrefs;
struct  pnode *yypref;
struct  pnode *yyprefs;
struct  tnode *tnd_alc   tndtab ;
struct  pnode *pnd_alc   pndtab ;
struct  pnode *cur_proc  NULL ;
struct  pnode *mod_name  NULL ;
struct  dnode *dnd_alc   dndtab ;

int     membuf [ MEMSIZE ];
int     mem_alc   30  ;
int     const_alc 0 ;
int     temp_alc  19 ;
int     yydebug  0 ;
int     yystate;
int     yychar ;
int     yynerrs ;
int     yyerrflag ;
int     yyline  1 ;
int     do_lev  0;
char    idbuf [ IDLEN ] ;
char    *idptr idbuf ;
char    constbuf [CONSTLEN] ;
char    *constptr  constbuf ;
char    textbuf [ TEXTLEN ];
char    *text_ptr   textbuf ;
char    pgm_type 0 ;
char    var_type 0;
char    do_type;
char    var_dim  0 ;
char    mem_type 0;
char    lexst 1;
char    pgmflag 0;
char    errflag  0;
char    cmtflag 0;
char    calflag 0;
char    casflag 0;
char    cascnt 0;
char    casexp;
char    exp_t;
char    rsflag 0;
char    labflag 0;
```

```
char     parmcnt ;
char     dclflag 0;
char     addflag 1;
char     strflag 0;
char     fdflag 0;
char     dstflag 0;
char     prmflag  0 ;
char     staflag  1;
char     whlflag  0;
char     rpflag  0;
char     doflag 0;
char     m_tmp;
char     *labword;
struct   donode *lb_tmp;
struct   donode *lb;
struct   lnode   *ls;
struct   lnode   *ls_tmp;
struct   tnode   *t_tmp;
struct   lnode   *lstmp;
struct   pnode   *p_tmp;
struct   pnode   *ptmp;
struct   dnode   *d_tmp;
int      t_type;
int      intr_tmp ;
int      cond;
int      cin;
*
```

```
#define MEMSIZE  100
#define MAXLINE  150
#define MAXDEPTH 150
#define TNDSIZE  50
#define PNDSIZE  10
#define DNDSIZE  50
#define DOBKSZ   25
#define DCLSIZE  50
#define LABSTSZ  15
#define IDLEN    10
#define CONSTLEN 10
#define TEXTLEN  100
#define NULL     0
#define ON       1
#define YES      1
#define NO       0
#define OFF      0

/*****************************************************************
 ***      EXTERNAL DECLARATIONS OF VARIABLES IN PINYIN      ***
 *****************************************************************/
struct intrnode {
       int opcode;
       int opr1;
       int opr1_t;
       int opr2;
       int opr2_t;
       int result;
       int result_t; } ;

struct lnode {
       char *lsword;
       int intr_ptr;
       struct lnode *bck_link;
       struct lnode *lnd_link;} ;

struct dnode {
       int dtype;
       int dim;
       struct dnode *dlink;
       int mem_ptr;
       int mem_t;
       char *s_ptr; } ;

struct tnode {
       char *tword;
       struct dnode *dnd_ptr;
       struct tnode *left;
       struct tnode *lf_back;
       struct tnode *right;
       struct tnode *rg_back; } ;
```

```
struct dclnode {
       struct tnode *tptr;
       struct dclnode *next; } ;

struct donode {
       char *lword;
       struct dclnode *dcl_link;
       struct lnode *lsta;
       struct lnode *gosta;
       struct donode *up_lev;
       int dotype; /* x0000 - simple do block */
                   /* x1000 - DO case block   */
                   /* x2000 - do iterate block*/
                   /* x3000 - do while block  */
       int inx_mem;
       char mt;
       int intr_p;
       int delt; } ;

struct pnode {
       char *pword;
       int ptype;
       struct tnode *locvar;
       struct tnode *form_var;
       int intrpnt;
       struct pnode *child;
       struct pnode *subling;
       struct donode *doptr;
       struct donode *cur_do;
       struct lnode *lsptr;
       struct lnode *gsptr;
       struct pnode *ret_addr;}  ;

struct parsnode {
       int s;
       int v;
       struct tnode *tref;
       struct pnode *pref;
       int           mref;
       char         *sref;
       int           mref_t; } ;

extern struct intrnode  intrtab[];
extern struct lnode     lstab[];
extern struct dnode     dndtab[];
extern struct tnode     tndtab[];
extern struct dclnode   dcltab[];
extern struct donode    dotab[];
extern struct pnode     pndtab[];
extern struct parsnode  parstab[];
```

```
extern struct parsnode  *pars_p;
extern struct donode    *do_alc;
extern struct dclnode   *dcl_alc;
extern struct lnode     *l_alc;
extern struct tnode     *yytref;
extern struct tnode     *yytrefs;
extern struct pnode     *yypref;
extern struct pnode     *yyprefs;
extern struct tnode     *tnd_alc;
extern struct pnode     *pnd_alc;
extern struct pnode     *cur_proc;
extern struct pnode     *mod_name;
extern struct dnode     *dnd_alc;
extern struct donode    *lb_tmp;
extern struct donode    *lb;
extern struct lnode     *ls;
extern struct lnode     *ls_tmp;
extern struct tnode     *t_tmp;
extern struct lnode     *lstmp;
extern struct pnode     *p_tmp;
extern struct pnode     *ptmp;
extern struct dnode     *d_tmp;
extern int    retstk[];
extern int    *rstkptr ;
extern int    intralc;
extern int    intrstk [];
extern int    istkptr ;
extern int    tmpstk [];
extern int    tstkptr ;
extern int    yymref;
extern int    yymrefs;
extern int    yym_t;
extern int    yyline;
extern char   *yysref;
extern char   *yysrefs;
extern int    membuf [];
extern int    mem_alc;
extern int    const_alc;
extern int    temp_alc;
extern int    yydebug;
extern int    yystate;
extern int    yychar;
extern int    yynerrs;
extern int    yyerrflag;
extern int    do_lev;
extern char   idbuf [];
extern char   *idptr ;
extern char   constbuf[];
extern char   *constptr;
extern char   *textbuf [];
extern char   *text_ptr;
```

```
extern char   pgm_type;
extern char   var_type;
extern char   do_type;
extern char   var_dim;
extern char   mem_type;
extern char   lexst;
extern char   pgmflag;
extern char   errflag;
extern char   cmtflag;
extern char   calflag;
extern char   casflag ;
extern char   cascnt;
extern char   casexp;
extern char   exp_t;
extern char   rsflag ;
extern char   labflag;
extern char   parmcnt;
extern char   dclflag;
extern char   addflag;
extern char   strflag;
extern char   fdflag;
extern char   dstflag;
extern char   prmflag;
extern char   staflag;
extern char   whlflag;
extern char   rpflag;
extern char   doflag;
extern char   m_tmp;
extern char   *labword;
extern int    t_type;
extern int    intr_tmp;
extern int    cond;
extern int    cin;
%
```

```
/*************************************************************************
 **                                                                     **
 **              ****      GRAMMAR  RULES     ****                     **
 **        A context-free language is used for the grammar              **
 **        rules of PINYIN.  Fifty-three terminals, 73 non-             **
 **        terminals, and 151 grammar rules are used in this            **
 **        grammar set.                                                 **
 **        The action code is written in C and is activated             **
 **        only when the structure is recognized.  The left-            **
 **        hand side of the first grammar rule is called the            **
 **        start symbol.  In effect, the parser is designed             **
 **        to recognize the start symbol.                               **
 **        Grammar rules of PINYIN are based on PL/M grammar.           **
 **        To make an easy process and avoid ambiguous usage            **
 **        several grammar rules are modified. Therefore, some          **
 **        features used in PL/M are not included or some features      **
 **        are modified.                                                **
 **        Upon giving the grammar rules YACC produces LALR(1)          **
 **        states and several large tables and writes them              **
 **        into a file called "y.tab.c."                                **
 **                                                                     **
 *************************************************************************/

%token   ADDRESS  AND  AT  BASED  BY  BYTE  CALL  CASE  DECLARE
         DATA  DO  ELSE  END  EOF  GO  GOTO  ID  IF  INITIAL INPUT
         MOD  NUMBER    OR OUTPUT  PROCEDURE  RETURN  STRING
         THEN  TO  WHILE
%left    NOT
%left    '+'  '-'
%left    '*'  '/'  MOD
%left    UMINUS
%{
#include  "m1.c"
#include  "def.c"
%}

%%

    /*************************************/
    /*      grammar rules section        */
    /*************************************/

program:                /*  program is the start symbol  */
                        module
       |                module  EOF  ;

module:                 module.name  simple.do.block  ;

module.name:            label.definition  =
```

```
                        {
                        mod_name = yypref;    };

label.definition:       ID  ':'  =
                        {
                        if (mod_name != NULL )
                        {  if (yytref->dnd_ptr == NULL )
                            deltnode();
                           else
                            errout ( 1, yytref->tword);
                           labflag = ON ;
                           labword = yytref ->tword;
                           lslink();
                           p_tmp = cur_proc ;
                           ckgoall (p_tmp);
                        }   };

simple.do.state.head : simple.do.statement  decpart  =
                        {
                        dclflag = OFF;
                        dstflag = OFF;    };

simple.do.block:        simple.do.state.head   unit ending
                |       simple.do.state.head   ending
                |       simple.do.statement  ending
                |       simple.do.statement  unit  ending  ;

unit:                   statement
                |       unit  statement
                |       unit  error  ';' =
                        {
                        prserr();             };

statement    :          basic.statement =
                        {
                        clrtmp();
                        chkcase();            }
                |       if.statement    =
                        {
                        clrtmp();
                        chkcase();         }
                |       label.definition  statement  ;

    decpart  :          declaration
                |       decpart   declaration  ;

declaration  :          declaration.statement  ';'
                |       procedure.definition  ;

basic.statement:        assignment    ';'
                |       assignment   error  ';'  =
```

```
                    {
                    prserr();
                    printf (" probably missing ';'\n"); }
                |   return.statement
                |   call.statement
                |   input.statement   ';'
                |   input.statement   error  ';'  =
                    {
                    prserr();
                    printf(" probably missing ';'\n");  }
                |   output.statement  ';'
                |   output.statement  error  =
                    {
                    prserr();
                    printf(" probably missing ';'\n");  }
                |   do.block
                |   goto.statement ';'
                |   goto.statement  error  ';' =
                    {
                    prserr();   };

if.statement:       if.clause  statement  =
                    {
                    m_tmp = popintr();
                    intrtab [m_tmp].result = intralc + 1;  }
                |   if.clause  true.part  statement  =
                    {
                    m_tmp = popintr ();
                    intrtab [m_tmp].result = intralc + 1;  };

if.clause:          IF   expression    THEN   =
                    {
                    yyitalloc ( 15,pars_p[2].mref,pars_p[2].mref_t,NULL,
                           NULL,NULL,NULL);
                    pushintr();       }
                |   IF  expression  error  =
                    {
                    prserr();
                    printf (" probably missing  keyword THEN \n");  };

true.part:          basic.statement   ELSE  =
                    {
                    m_tmp = intrstk[istkptr] ;
                    intrtab [m_tmp].result = intralc + 2 ;
                    yyitalloc(13,NULL,NULL,NULL,NULL,NULL,NULL);
                    intrstk[istkptr] = intralc ;   };

group:              group.head   ending  ;

group.head:         DO   step.definition
                |   DO   while.clause ';'
```

```
                |   DO   case.selector  ';'   =
                    {
                    staflag = ON ;   }
                |   group.head  statement    ;

do.block:           group
                |   simple.do.block   ;

step.definition:    simple.variable   replace   expression iteration.control =
                    {
                    lb_tmp = cur_proc->cur_do ;
                    lb_tmp->dotype = lb_tmp->dotype | 02000 ;
                    lb_tmp->inx_mem = yymref ;
                    yyitalloc(14,yymref,NULL,pars_p[3].mref,pars_p[3].mref_t,
                          NULL,NULL);
                    intr_tmp = tmpalloc();
                    yyitalloc(10,yymref,NULL,pars_p[4].mref,pars_p[4].mref_t,
                          intr_tmp,NULL);
                    pushintr();
                    yyitalloc(15,temp_alc,NULL,NULL,NULL,NULL,NULL);
                    pushintr();
                    staflag = ON ;
                    dclflag = OFF ;    };

iteration.control:  TO  expression  ';'  =
                    {
                    lb_tmp = cur_proc->cur_do ;
                    do_type = lb_tmp->dotype;
                    m_tmp = ckconst (1);
                    do_type = do_type | m_tmp ;
                    lb_tmp->dotype = do_type;
                    yymref = pars_p[2].mref;
                    yym_t  = pars_p[2].mref_t;  }
                |   TO  expression  error  ';'  =
                    {
                    prserr();
                    printf("    probably missing ';'\n");  }
                |   TO expression  BY  expression  error ';'  =
                    {
                    prserr();
                    printf(" probably missing ';'\n");  }
                |   TO  expression  BY  expression  ';'  =
                    {
                    lb_tmp = cur_proc->cur_do ;
                    do_type = lb_tmp->dotype ;
                    if (pars_p[4].mref_t == 2)
                       lb_tmp->mt = 2;
                    do_type = do_type | pars_p[4].mref;
                    lb_tmp->dotype = do_type;
                    yymref = pars_p[2].mref;
                    yym_t = pars_p[2].mref_t ;  };
```

```
simple.do.statement:  DO   ';' =
                      {
                      if  (casflag ) doflag = ON ;
                        dclflag = ON;  };

while.clause:         WHILE  expression  =
                      {
                      yyitalloc(15,pars_p[2].mref,pars_p[2].mref_t,NULL,NULL,
                            NULL,NULL);
                      pushintr();   };

case.selector:        CASE  expression  =
                      {
                      casexp = pars_p [2].mref ;
                      exp_t = pars_p[2].mref_t ;
                      cascnt = 0;
                      caselector ();   };

procedure.head:       procedure.name   ';'
                    | procedure.name  basic.type  ';'
                    | procedure.name  parameter.list  ';'
                    | procedure.name  parameter.list  basic.type  ';'  ;

procedure.name.head:  PROCEDURE ;

procedure.name:       procedure.name.head  ID =
                      {
                      yypref = pars_p [2].pref ;
                      yysref = yypref->pword;
                      pgmflag = OFF ;
                      prmflag = ON;
                      if ( cur_proc->child != NULL)
                      {
                        ptmp = cur_proc ->child;
                        while ( ptmp->subling != NULL)
                           ptmp = ptmp->subling ;
                        ptmp->subling = yypref ;
                      }
                      else
                         cur_proc->child = yypref ;
                      yypref->ret_addr = cur_proc ;
                      cur_proc = yypref ;
                      yytref = NULL;   };

procedure.definition:  procedure.head  decpart  unit  ending
                     | procedure.head  unit  ending
                     | procedure.head  decpart  ending
                     | procedure.head  ending  ;

parameter.list:       parameter.head  variable   ')'  =
```

```
                    {
                    chkprm ();
                    offcpfg();  }
                |   parameter.head   locator  ')' =
                    {
                    prsprm(0);
                    offcpfg();   }
                |   parameter.head  constant  ')'  =
                    {
                    prsprm(1);
                    offcpfg();   };

parameter.head:     '('
                |   parameter.head  variable   ',' =
                    {
                    chkprm();  }

                |   parameter.head  locator  ',' =
                    {
                    prsprm(0);  }
                |   parameter.head  constant  ',' =
                    {
                    prsprm(1);  };

ending:             END   ';' =
                    {
                    lb_tmp = cur_proc->cur_do ;
                    labword = NULL;
                    if ( lb_tmp->delt  >> 6 == 1 )
                    {
                       cur_proc = cur_proc->ret_addr ;
                       if (cur_proc == NULL )
                         errout ( 3, NULL );
                    }
                    else
                        closedo ();  }

                |   END   ID    ';'  =
                    {
                    yytref = pars_p[2].tref;
                    if (yytref->dnd_ptr == NULL )
                    {  deltnode();
                       labword = yytref->tword; }
                    else
                    {     errout ( 1, yytref->tword );
                          labword = NULL ; }
                    lb_tmp = cur_proc ->cur_do ;
                    if ( lb_tmp->delt >> 6 == 1 )
                    {  if ( cur_proc->ret_addr == NULL)
                          errout ( 3, NULL );
                       else
```

```
                    {  if ( labword != NULL )
                       { if (( cond=strcmp(labword,cur_proc->pword))
                            != 0 )
                           errout ( 11, labword );
                        }
                       cur_proc = cur_proc->ret_addr ;
                    }
                 }
                 else closedo();  }
            |    label.definition  ending  ;

return.statement:    typed.return
            |    untyped.return   ;

typed.return:        RETURN   expression  ';'  =
                     {
                     yymref = pars_p [2].mref ;
                     yym_t  = pars_p [2].mref_t ;
                     retpush (0) ; };

untyped.return:      RETURN  ';'  =
                     {
                     retpush (1) ;  };

call.statement:      call.head ';' =
                     {
                     calpush();  }
            |    call.head parameter.list ';'    =
                     {
                     calpush();  };

call.head     :      CALL ID  =
                     {
                     yytref = pars_p[2].tref;
                     labword = yytref-> tword;
                     if ( !fdflag)
                     {  deltnode();
                        chkproc ();
                        if ( !fdflag)
                          errout ( 4, labword);
                        else
                          yymref = yypref->intrpnt;
                     }
                     else
                     errout ( 4, labword); };

goto.statement:      goto  ID  =
                     {
                     intr_tmp = MAXLINE;
                     yytref = pars_p[2].tref;
                     labword = yytref ->tword;
```

```
                    if ( yytref ->dnd_ptr == NULL )
                    deltnode ();
                    else
                       errout ( 1,labword);
                    p_tmp = cur_proc ;
                    fdlab ( p_tmp);
                      if ( intr_tmp == MAXLINE)
                      {  lb_tmp = cur_proc ->cur_do;
                         ls_tmp = lb_tmp->gosta;
                         ls= lb_tmp->gosta = yylsalloc();
                         ls->lnd_link = ls_tmp;
                         if ( ls_tmp != NULL )
                              ls_tmp->bck_link = ls;
                         ls->lsword = labword ;
                         ls->intr_ptr = intralc;  }
                      else
                         intrtab [intralc].result = intr_tmp; };

goto:                    GO  TO  =
                         {
                         yyitalloc(13,NULL,NULL,NULL,NULL,NULL,NULL); }
                  |      GOTO   =
                         {
                         yyitalloc(13,NULL,NULL,NULL,NULL,NULL,NULL); };

declaration.statement:   DECLARE  declaration.element =
                         {
                         clrdecl ();  }

                       | declaration.statement   ',' declaration.element  =
                         {
                         clrdecl ();  };

declaration.element:     type.declaration
                       | ID  data.list  =
                         {
                         d_tmp = yydalloc(yytref);
                         d_tmp->dtype = var_type;
                         m_tmp = d_tmp->mem_ptr;
                         membuf [m_tmp] = pars_p[2].v ;  };

data.list:               data.head  NUMBER  ')'  ;
data.head:               DATA  '(' =
                         {
                         var_type = var_type  |  0003; }


                       |  data.head  NUMBER   ','  ;

type.declaration:        ID   type  =
                         {
```

```
                    d_tmp = yydalloc(yytref);
                    d_tmp->dtype = var_type;
                    ckvart(0);  }
                |   array.specifier   type  =
                    {
                    d_tmp = yytref->dnd_ptr ;
                    d_tmp->dtype = var_type;
                    ckvart (1);  }
                |   type.declaration  initial.list    =
                    {
                    if ( yym_t & 0007 == 2 )
                    {
                     errout ( 11,yytref->tword);
                     break; }
                    else
                    {
                     for ( cond= 0; rstkptr >=retstk; cond++ )
                     {
                        membuf [yymref+cond] = retstk[cond] ;
                        rvltype( yymref+cond, yym_t);
                        rstkptr -- ;  }
                    }
                    rstkptr = &retstk[0] -1 ;  };

array.specifier:    array.head   NUMBER  ']'  =
                    {
                    d_tmp = yytref->dnd_ptr ;
                    d_tmp->dim = pars_p [2].v ;
                    mem_alc =+  pars_p [2].v -1 ;
                    yymref = d_tmp->mem_ptr; }
                |   array.head    '*'      ']'  ;

array.head:         ID  '['  =
                    {
                    d_tmp = yydalloc(yytref);
                    yymref = d_tmp->mem_ptr ; };

type:               variable.type
                |   based.variable  basic.type  ;


variable.type1:     basic.type  ;

variable.type2 :    variable.type1  AT  '('  =
                    {
                    var_type = var_type | 0001 ; };

variable.type :     variable.type1
                |   variable.type2   restr.expression  ')'  =
                    {
                    yymref = pars_p [2].mref; };
```

```
basic.type:             BYTE  =
                        {
                        var_type = var_type ! 0100 ; }
                      ! ADDRESS  =
                        {
                        var_type = var_type ! 0000; };

based.variable:         BASED  ID =
                        {
                        var_type = var_type ! 0002 ;
                        yytref = pars_p [2].tref;
                        labword = yytref->tword;
                        p_tmp = cur_proc;
                        ckdecl (0);
                        d_tmp = yytref->dnd_ptr ;
                        t_type = d_tmp->dtype;
                        if (( t_type >> 6 ) != 0  ||
                           (t_type & 0001 ) != 0 )
                        {
                           errout (7, labword);
                           d_tmp->dtype = 0000;
                        }
                        yymref = d_tmp->mem_ptr ; };

initial.list:           initial.head  NUMBER  ')'  =
                        {
                        if ( rstkptr < retstk + 19 )
                        {
                           rstkptr ++ ;
                           *rstkptr = pars_p[2].v ;
                        } };

initial.head:           INITIAL  '('  =
                        {
                        rstkptr = &retstk[0] -1 ; }
                      ! initial.head  NUMBER  ','  =
                        {
                        if ( rstkptr < retstk + 19 )
                        {
                           rstkptr ++;
                           *rstkptr = pars_p[2].v ; } };

assignment:             variable  replace   expression =
                        {
                        yyitalloc(14,yymref,yym_t,pars_p[3].mref,pars_p[3].mref_t,
                             NULL,NULL);
                        ckaddr (); }
                      ! left.part  assignment   =
                        {
                        yyitalloc(14,yymref,yym t,intrtab[intralc].oor2.
```

```
                        intrtab[intralc].opr2_t,NULL,NULL);
                    ckaddr ();  };
input.statement:    variable  replace  INPUT  =
                    {
                    yyitalloc(26,yymref,yym_t,0,0,0,0 );  };

output.statement:   OUTPUT   replace  variable  =
                    {
                    yymref = pars_p[3].mref ;
                    yym_t = pars_p[3].mref_t;
                    yyitalloc(27,yymref,yym_t,0,0,yyline,0); };

replace:            '='  ;

left.part:          variable  ','  ;

expression:         logical.expression
                  | restr.expression  =
                    {
                    rsflag = ON ;  };

locator:            '.'  ID  =
                    {
                    yytref = pars_p [2].tref ;
                    yysref = pars_p [2].sref;
                    labword = yytref->tword;
                    p_tmp = cur_proc;
                    ckdecl (1);
                    d_tmp = yytref->dnd_ptr;
                    yymref = d_tmp->mem_ptr ;
                    yym_t = 1;   };

restr.expression:   locator  '+'  NUMBER  =
                    {
                    if (dstflag)
                       yymref = yymref + pars_p [3].v;
                    else
                       locpush (1) ;  }
                  | locator  '-'  NUMBER  =
                    {
                    if (dstflag)
                         yymref = yymref + pars_p [3].v ;
                    else
                         locpush (2) ; }
                  | locator ;

logical.expression: logical.factor
                  | logical.expression  OR   logical.factor   =
                    {
                    compush (5); };
```

```
logical.factor:         logical.secondary
                    |   logical.factor AND  logical.secondary  =
                        {
                        compush (6);  };

logical.secondary:      logical.primary
                    |   NOT  logical.primary  =
                        {
                        intr_tmp = tmpalloc();
                        yyitalloc(20,pars_p[2].mref,pars_p[2].mref_t,NULL,NULL,
                            intr_tmp,NULL);
                        yymref = temp_alc;
                        yytref = NULL ;  };

logical.primary:        arith.expression
                    |   arith.expression  relation  arith.expression  =
                        {
                        compush (pars_p[2].v) ; };

relation:               '==' =
                        {
                         yyval = 9; }
                    |    '<' =
                        {
                         yyval = 7 ;  }
                    |    '>' =
                        {
                         yyval = 8 ;  }
                    |    com   ;

com:                    '<=' =
                        {
                         yyval = 10; }
                    |    '>=' =
                        {
                         yyval = 11; }
                    |    '<>' =
                        {
                         yyval = 12 ;  };

arith.expression:       term
                    |   arith.expression  '+'  term  =
                        {
                        compush (1) ; }
                    |   arith.expression  '-'  term  =
                        {
                        compush (2); };

term:                   primary
                    |   term  '*'  primary  =
                        {
```

```
                    compush (3); }
                |   term  '/'  primary =
                    {
                    compush (4); }
                |   term  MOD  primary  =
                    {
                    compush (16) ; };

primary:            constant
                |   variable
                |   sub.expression
                |   '-'  primary  %prec  UMINUS     =
                    {
                    intr_tmp = tmpalloc();
                    yyitalloc(21,pars_p[2].mref,pars_p[2].mref_t,NULL,NULL,
                        intr_tmp,NULL); };

sub.expression:     '('  expression  ')'   =
                    {
                    yymref = pars_p [2].mref;
                    yym_t = pars_p [2].mref_t; };

variable:           simple.variable
                |   subscript.head  expression  ')'  =
                    {
                    tmpalloc();
                    intr_tmp = yym_t  |  1;
                    yyitalloc(1,yymref,intr_tmp,pars_p[2].mref,pars_p[2].mref_t,
                        temp_alc,2);
                    yymref = temp_alc ;
                    yym_t = yym_t | 2 ;  };

constant:           NUMBER  =
                    {
                    chconst();  }
                |   STRING  ;

subscript.head:     simple.variable  '('  ;

simple.variable:    ID =
                    {
                    labword = yytref ->tword;
                    p_tmp = cur_proc ;
                    if ( prmflag ) break;
                       ckdecl (1);
                    d_tmp = yytref->dnd_ptr;
                    yymref = d_tmp->mem_ptr ;
                    yym_t = d_tmp->dtype ;  };
```

```
#include  "m1.c"
#include  "def.c"
extern int yyval;
extern int yylval ;
extern fin, fout;
/*************************************************************************
 **                                                                     **
 **                *===*     PARSER & ERROR HANDLER     *===*          **
 **    By giving the grammar rules YACC builds a bottom-up LALR(1)      **
 **    parser.  The LALR(1) parser consists of two parts: the driver    **
 **    routine and the parsing table.  The parsing table is giving in   **
 **    the file "y.tab.c."   There are four kind of actions to be taken **
 **    while parsing.  They are :                                       **
 **    1.  ERROR : An improper token is found on the input stream.      **
 **    2.   TEST : Tests for an appropriate token on the input stream.  **
 **    3.  SHIFT : Shifts to next specified state & stacks input token. **
 **    4. REDUCE : Reduces by the appropriate grammar rule.             **
 **    5. ACCEPT : Accepts this input stream.                           **
 **                                                                     **
 **    Errors can be encountered in all of the phases of PINYIN.  The   **
 **    token name, "error", is reserved for error handling.  It         **
 **    suggests places where errors are expected and recovery might     **
 **    take place.  To prevent a cascade of error message, the parser   **
 **    assumes that, after detecting an error, it remains in error      **
 **    state until three tokens are successfully read and shifted.      **
 **    If an error is detected when the parser is already in error      **
 **    state, no error message is produced.                             **
 **                                                                     **
 *************************************************************************/
main(argc, argv)
int argc;
char *argv[];
{
cin = copen ( argv[1], 'r' );
if ( cin < 0 ) printf ( " cannot open input ");
        if ( yyparse () == 0  && !errflag )
            yyaccept ();
        else printf (" please correct errors and try it again !!\n");
}
copen(s,c)
char *s;
char c;
{
int f;
        if ( c == 'r' )
{
        f = fin = open ( s, 0);
}
return ( f);
}
```

```
/*********************************************
 ***            PARSER                     ***
 *********************************************/
yyparse()
{
extern int yygo[], yypgo[], yyr1[], yyr2[], yyact[], yypact[];
auto int ac, n, *p ;
        yystate = 0;
        yychar  = -1;
        yynerrs = 0;
        yyerrflag = 0;
        pars_p = &parstab[0] -1 ;
stack:  /* put a state and value onto the stack */
        if (yydebug)
            printf ("state %d,value %d, mref %d\n",yystate,yyval,yymref);
        ++pars_p;
        pars_p->s = yystate;
        pars_p ->v = yyval;
        pars_p ->tref = yytref;
        pars_p ->pref = yypref;
        pars_p ->mref = yymref;
        pars_p ->mref_t = yym_t ;
        pars_p ->sref = yysref ;

newstate:       /* set p to point to the parsing actions for the new state */
        p = &yyact [ yypact[yystate+1] ] ;
actn:           /* get the next action, and perform it */
        n = (ac = *p++ ) & 07777; /* n is the "address" of the action */
        switch ( ac>>12) /* switch on operation */
        {
        case  1: /* skip on test */
                if ( yydebug && (yychar <0 ) )
                {    yychar = yylex () ;
                     printf (" character %d read \n",yychar );
                }
                if ( n != ((yychar <0 ) ? (yychar= yylex() ) : yychar ))
                        ++p ;
                goto actn ; /* get next action */

        case  2: /* shift */
                yystate = n ;
                yyval = yylval ;
                yychar = -1 ;
                yypref = yyprefs;
                yytref = yytrefs;
                yysref = yysrefs;
                yymref = yymrefs;
                if (yyerrflag ) --yyerrflag ;
                goto stack ; /*stack new state */
        case  3: /* reduce */
                if (yydebug)
```

```
            printf ("reduce %d\n",n);
          pars_p =- yyr2 [n] ;
          yyval = pars_p [1].v ;
          yysref = pars_p [1].sref ;
          yymref = pars_p [1].mref;
          yym_t  = pars_p [1].mref_t ;
          yytref = pars_p [1].tref;
          yypref = pars_p [1].pref ;
          yyactr(n) ;
          /* consult goto table to find next state */
for ( p= &yygo[yypgo[yyr1[n]]]; *p != pars_p->s && *p >=0; p=+ 2 );
          yystate = p [1] ;
          goto stack ; /* stack new state and value */
case  4: /* accept */
          return (0) ;
case  0: /* error ...attempt to resume parsing */
          switch (yyerrflag)
          {
          case  0:  /* brand new error */
                  ++yynerrs ;
                  yyerror ("syntax error ") ;
          case  1:
          case  2: /* incompletely recovered error...try again */
                  yyerrflag = 3 ;
                  /* find a state where "error" is a legal shift action*/
                  while ( pars_p >= parstab )
                  {    for ( p= &yyact[ yypact [(pars_p->s)+1] ];
                          (*p>>12) == 1; p =+ 2 ) /* search action */
                              if ( *p == 4352 ) goto found ;
                          /* the current pars_p has no shift on "error", */
                          /*  pop stack */
                          if (yydebug)
                             printf ("error recovery pops state %d,uncovers %d\n",
                              pars_p->s, pars_p[-1].s);
                              --pars_p ;
                          }
                  /* there is no state on the stack with an error */
                  /* shift...abort */
          abort:
                  return (1);
          found: /* we have a state with a shift on "error", resume */
                  /* parsing */
                  yystate = p[1] & 07777 ;
                  goto stack ;
          case 3: /* no shift yet; clobber input char */
                  if (yydebug)
                  printf ("error recovery discards char %d\n",yychar);
                  if ( yychar == 0 ) goto abort ;/*do not discard EOF, */
                  yychar = -1 ;
                  goto newstate; /* try again in the same state */
          }
```

```
        }
}

/***********************************************
 ***           ERROR   HANDLER                ***
 ***********************************************/

yyerror (s)
char *s ;
{

extern  char  *yysterm[] ;
        if ( !errflag )
        {
          errflag = ON ;
          printf (" compile-time error messages :\n");
        }
        printf (" %s",s);
        if (yyline)
            printf (", line %d,", yyline);
        printf (" on input: ");
        if (yychar >= 0400 )
            printf ("%s\n",yysterm[yychar -0400] );
        else switch ( yychar )
             {
               case '\t':  printf ("\\t\n" ); return ;
               case '\n':  printf ("\\n\n" ); return ;
               case '\0':  printf ("$end\n"); return ;
               default   :  print ("%c\n",yychar); return ;
             }
}

errout(c,s)
char c;
char *s;
{
        if ( !errflag )
        {
           errflag = ON;
           printf (" compile-time error messages :\n");
        }

        switch (c)
        {
                case  1:
                        printf (" invalid label, line %d, on input: %s",
                                yyline, s);
                        break;
                case  2:
                        printf (" undefined variable, line %d, on input: %s\n",
                                yyline, s);
```

```
                break;
        case  3:
                printf (" extra END statement , line %d\n",yyline);
                break;
        case  4:
                printf(" invalid procedure name , line %d, on input: %s\n",
                        yyline,s);
                break;
        case  5:
                printf (" undefined parameter , line %d, on input: %s\n",
                          yyline, s);
                break;
        case  6:
                printf(" unmatched parameter type , line %d, on input: %s\n",
                           yyline, s);
                break;
        case  7:
                printf(" invalid variable type, line %d, on input : %s\n",
                           yyline, s);
                break;
        case  8: printf ("   %d   illegal character \n",yyline);
                break;
        case  9:
                printf(" symbol table overflow, line %d\n",yyline);
                break;
        case 10:
                printf (" memory overflow, line %d\n",yyline);
                break;
        case 11:
                printf(" invalid initialization, line %d\n",yyline);
                break;
    }
    return;
}
%
```

```
#include  "m1.c"
#include  "def.c"
#define  yyclearin yychar = -1
#define  yyerrok  yyerrflag = 0
extern int yylval;
extern int yyval;
/*************************************************************************
 **                                                                     **
 **                ****     LEXICAL  ANALYZER     ****                **
 **      The lexical analyzer separates characters of the input         **
 **      stream into tokens.  Fifty-three tokens can be recognized      **
 **      by the lexical analyzer.  The lexical analyzer is an           **
 **      integer function called "yylex."  The function returns         **
 **      an integer value which represents the type of the token.       **
 **      The type number of each token is chosen by YACC.  The          **
 **      type number is defined by using the "#define" mechanism        **
 **      in the file "y.tab.c."   There are 55 ASCII characters         **
 **      could be recognized by the lexical analyzer.                   **
 **                                                                     **
 *************************************************************************/
int  lex_act [27] [21] {
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
      1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
      4, 1, 1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
      5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
      6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
      7, 7, 7, 7, 7, 7, 7, 7,28, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
      8, 8, 8, 8, 8, 8, 8,27, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
      9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
     10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,
     11,11,11,11,11,11,11,11,11,11,11,11, 1, 1,11,11,11,11,11,11,11,
     12,12,12,12,12,12,12,12,12,12,12,12, 1,12,12,12,12,12,12,12,12,
     13,13,13,13,13,13,13,13,13,13,13,13, 1,13,13,13,13,13,13,13,13,
     14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,
     15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,
     16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,
     17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,17,
     18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,18,
     19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,19,
     20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,
     21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,
     22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,
     23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,23,
     24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,24,
     25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
     26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26, 0 };
```

```
int  lex_sta [27] [21] {
      2, 3,22,22, 0, 0,22,22,22,22,22,22,22,22,22,22,22,22,22,22,22,
      2, 3,22, 1, 1, 4, 5, 6, 7, 8, 9,10,11,12,17,18,19,20,21,24,25,
      2, 2, 2, 1, 1, 4, 5, 6, 7, 8, 9,10,11,12,17,18,19,20,21,24,22,
     22, 3,22, 1, 1, 4, 5, 6, 7, 8, 9,10,11,12,22,18,22,20,22,22,22,
     22, 3,22, 1, 1,23,23, 6, 7,22,22,23,23,23,17,22,22,22,22,22,25,
      2, 3,22, 1, 1,23,23, 6, 7,22,22,23,23,23,17,22,22,22,22,22,25,
      2, 3,22, 1, 1,23,23, 6, 1,22,22,23,23,23,17,22,22,22,22,22,25,
      2, 3,22, 1, 1,23,23, 1, 7,22,22,23,23,23,17,22,22,22,22,22,25,
      2, 3,22, 1, 1,22,22, 6, 7,22,22,22,22,22,22,22,22,22,22,22,22,
      2, 3,22, 1, 1,22,22, 6, 7,22,22,22,22,22,17,22,22,22,22,22,22,
      2, 3,22, 1, 1,23, 5, 6, 7,22,22,23,13,14,17,22,22,22,22,22,25,
      2, 3,22, 1, 1,23, 5, 6, 7,22,22,23,15,23,17,22,22,22,21,22,25,
      2, 3,22, 1, 1,23, 5, 6, 7,22,22,23,16,23,17,22,22,22,22,22,25,
      2, 3,22, 1, 1,23, 5, 6, 7,22,22,23,23,23,17,22,22,22,22,22,25,
      2, 3,22, 1, 1,23, 5, 6, 7,22,22,23,23,23,17,22,22,22,22,22,25,
      2, 3,22, 1, 1,23, 5, 6, 7,22,22,23,23,23,17,22,22,22,22,22,25,
      2, 3,22, 1, 1,23, 5, 6, 7,22,22,23,23,23,17,22,22,22,22,22,25,
      2, 3,22, 1, 1,23, 5, 6, 7,22,22,23,23,23,17,22,22,22,21,22,25,
      2,22,22, 1, 1, 4, 5, 6, 7, 8, 9,10,11,12,22,18,22,20,22,22,22,
      2, 3,22, 1, 1,22,22, 6, 7,22,22,22,22,22,22,22,22,22,22,22,22,
      2, 3,22, 1, 1, 4, 5, 6, 7, 8, 9,10,11,12,22,18,22,22,21,22,22,
      2, 3,22, 1, 1,22,22, 6, 7,22,22,22,22,22,22,22,22,22,22,22,22,
      2, 3,22, 1, 1, 4, 5, 6, 7, 8, 9,10,11,12,17,18,19,20,21,22,25,
      2, 3,23, 1, 1,23,23, 6, 7,23,23,23,23,23,17,18,22,22,22,22,25,
      2,23,23, 1, 1,23,23, 6, 7,23,23,23,23,23,23,23,23,23,23,23,23,
     25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,25,26,
      2, 3,22, 1, 1, 4, 5, 6, 7, 8, 9,10,11,12,17,18,19,20,21,24,25 };

yylex()
{
        int c,col,indx,act ;
        struct tnode *root ;
        struct pnode *yypalloc() ;
        struct tnode *srchform();
        struct tnode *yybintree ();
        auto struct tnode *form_root;
        yylval = 0;
        yytref  = NULL ;
        yypref  = NULL ;
        yymref  = NULL ;
        yysref  = NULL ;
        yytrefs = NULL;
        yyprefs = NULL;
        yysrefs = NULL;
        yymrefs = NULL;
        yym_t = NULL ;
        while ( (c=getchar() ) !=NULL)
        {
             col = getcol(c) ;
```

```
act = lex_act [lexst] [col] ;
lexst = lex_sta [lexst] [col] ;
if ( ! cmtflag || act == 28 )
{
switch(act)
{
   case  0:   /* set on string flag */
          strflag  = ON ;
          break;
   case  1: break;
   case  2: /* turn off strflag */
          strflag = OFF ;
          break;
   case  3:  /*  return an ID or a keyword */
          *idptr = '\0' ;
          idptr = idbuf ;
          if (( indx = srchkey(idbuf)) == -1)
          {  if (mod_name == NULL || pgmflag )
             {   yypref = yypalloc();
                 yysref = yypref->pword;
                 if ( mod_name == NULL )
                    cur_proc = yypref;   }
             else {
                  form_root = cur_proc ->form_var;
                  if (prmflag == ON)
                  {  if (form_root == NULL)
                        yytref= cur_proc->form_var = yytalloc();
                     else
                     {  while (form_root->left != NULL)
                           form_root = form_root->left;
                        yytref= form_root->left= yytalloc();
                     }
                     yysref = yytref->tword;
                  }
                  else
                  {  /* search formal variable list first */
                     yytref = srchform(form_root,idbuf);
                     if (yytref == NULL)
                     {   root = cur_proc->locvar;
                         yytref = yybintree(root,idbuf);
                         if (root ==NULL)
                            cur_proc->locvar = yytref;
                     }
                     yysref = yytref->tword;
                  }
                 }
                  yytrefs = yytref;
                  yyprefs = yypref;
                  yysrefs = yysref;
                  return (ID);
          }
```

```
        else return (indx) ;
        break ;
case  4:  /* return a constant */
        *constptr = '\0' ;
        constptr = constbuf ;
        yysrefs = constbuf;
        yylval = atoi (constbuf);
        yymref = ckconst(yylval) ;
        yymrefs = yymref;
        return (NUMBER) ;
        break;
case  5:
        ckquote ();
        return ('+') ;
case  6:
        ckquote();
        return ('-');
case  7:
        ckquote();
        return ('*') ;
case  8:
        ckquote();
        return ('/') ;
case  9:
        return (',');
case 10:
        {
        prmflag = OFF ;
        return (';');
        }
case 11:
        ckquote();
        return ('<');
case 12:
        ckquote();
        return ('=') ;
case 13:
        ckquote();
        return ('>');
case 14:
        ckquote();
        return (290);
case 15:
        ckquote();
        return (292);
case 16:
        ckquote();
        return (289);
case 17:
        ckquote();
```

```
        return (291);
case 18:
        {
        ckquote();
        if ( calflag )
          parmcnt = 0;
        return ('(');
        }
case 19:
        return (')');
case 20:
        return ('[');
case 21:
        return (']');
case 22:
        return ('.');
case 23:
        ckquote();
        printf ("invalid character\n");
        idptr = idbuf ;
        constptr = constbuf ;
        break;
case 24:
        ckquote();
        printf ("two adjacent operand \n") ;
        break;
case 25:
        return (':');
        break;
case 26:  /* return a string */
         idptr [-1] = '\0' ;
         idptr = idbuf ;
         yysref = strsave (idbuf);
         return (STRING);
case 27:  /*  turn on comment flag  */
        cmtflag = ON ;
        break;
case 28:  /* turn off comment flag  */
        cmtflag = OFF ;
        break;
        }
}
}
return (NULL) ;
}
```

```
#include "a1.c"
#include  "def.c"
/*************************************************************************
 **      This file comprises some routines which support the        **
 **      semantic action.                                           **
 *************************************************************************/
/*=========================================================*/
/*               Checks next input character               */
/*=========================================================*/
getcol(c)
int c;
{
int pos;
char *alph;
char *digit;
char *spchar;

        alph = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" ;
        digit= "0123456789" ;
        spchar = "$ \n+-*/,;<=>()[].:''";
        if ((pos = yyindex (alph,c) ) >= 0)
        {     concat (c,'i') ;
              return (0) ;
        }
        else {
             if ((pos = yyindex (digit,c) ) >= 0)
                {     concat (c,'c');
                      return (1);
                }
                else {
                     if ((pos = yyindex (spchar,c) ) >= 0)
                        {  if ( strflag || (pos==0) )
                                concat(c,'i');
                           if ( pos == 2) /* line feed */
                              yyline ++ ;
                           return ( 2+pos);
                       }
                       else {
                               errout ( 8, NULL );
                           return (3);
                           }
                        }
                }
}

/*=========================================================*/
/*                Concatenates next input character        */
/*=========================================================*/
concat(c,t)
```

```
int  c;
char t;
{
        if ( cmtflag) return ;
        if (t == 'i' )
        {  if (idptr < idbuf + IDLEN  -1 )
                *idptr ++ = c;
        }
        else {
             if ( t== 'c' )
             {
                if ( (idptr > idbuf) || strflag )
                {  if (idptr < idbuf + IDLEN -1)
                        *idptr ++ = c;
                }
                else if (constptr < constbuf + CONSTLEN  -1 )
                        *constptr ++ = c;
                }
             }
}
/*=========================================================*/
/*              Allocates temporary storage                */
/*=========================================================*/

tmpalloc()
{

        if ( temp_alc < 29 )
        {  temp_alc ++ ;
           if ( staflag == ON )
           {  staflag = OFF ;
                tstkptr ++ ;
                tmpstk[tstkptr] = temp_alc ;
           }
        }
        return (temp_alc );
}

/*=========================================================*/
/*                  Resets declarations                    */
/*=========================================================*/

clrdecl()
{
        var_type = 0;
        var_dim = 0;
}

/*=========================================================*/
/*                  Resets temporary storage               */
/*=========================================================*/
```

```
clrtmp()
{
        if ( temp_alc > 19 && tstkptr > 0)
        {
        if ( tmpstk [tstkptr] != casexp )
        {
        temp_alc = tmpstk [tstkptr]  - 1;
        tstkptr -- ;
        }
        }
        staflag = ON ;
        labflag = OFF ;
        rsflag = OFF ;
        return;
}

/*============================================================*/
/*      Generates a sequence of quadruple statements for      */
/*      a case statement.                                     */
/*============================================================*/
caselector ()
{

        tmpalloc();
        yyitalloc(12,casexp,exp_t,cascnt,1,temp_alc,NULL);
        yyitalloc(15,temp_alc,NULL,NULL,NULL,NULL,NULL);
        pushintr () ;
        return ;
}

/*============================================================*/
/*        Checks each single statement in a DO CASE block.    */
/*============================================================*/
chkcase()
{

        if ( casflag == ON &&  !doflag )
        {
            m_tmp = popintr ();
            intrtab [m_tmp].result = intralc + 2;
            yyitalloc(13,NULL,NULL,NULL,NULL,NULL,NULL);
            pushintr () ;
            cascnt ++ ;
            caselector ();
            return ;
        }
        return ;
}

/*============================================================*/
/*       Searches a node in the symbol table.                 */
```

```
/*       Binary-search is used as a look up method.        */
/*=========================================================*/
struct tnode *yybintree( ptr,sym)
struct tnode *ptr ;
char *sym ;
{
struct tnode *t_ptr;


        fdflag = NO;
        if (ptr !=  NULL )
        {    while (( cond = strcmp(sym,ptr->tword) ) != 0 )
             {    if ( cond < 0 )
                  {
                        if ( ptr->left == NULL )
                        {  if (addflag)
                           {  ptr->left = yytalloc();
                              t_ptr = ptr->left ;
                              t_ptr ->lf_back = ptr ;
                              return ( ptr->left);
                           }
                           else return (NULL);
                        }
                        else ptr = ptr->left ;
                  }
                  else
                        {    if ( ptr->right == NULL )
                                {  if (addflag)
                                   {  ptr->right = yytalloc();
                                      t_ptr = ptr->right ;
                                      t_ptr -> rg_back = ptr ;
                                       return (ptr->right);
                                    }
                                    else return (NULL);
                                }
                             else ptr = ptr->right ;
                        }
             }
             fdflag = YES ;
             return (ptr);
        }
        else {
        ptr = yytalloc() ;
        return (ptr);
        }
}
```

```
/*=========================================================*/
/*     This procedure is involked when a goto statement is */
/*     reached.  It checks the exclusive extent of         */
/*     an enclosing block.                                 */
/*     lb_tmp is a pointer which points to the current do  */
/*     block when the procedure is active.                 */
/*=========================================================*/


chencls()
{


        while ( lb_tmp != NULL )
        {
           if (( cond = strcmp(labword,lb_tmp->lword)) !=0)
           {
              if ( lb_tmp->delt >> 6  == NO )
              {
                 ls_tmp = lb_tmp->lsta;
                while ( ls_tmp != NULL )
                 {
                    if ((cond = strcmp (labword,ls_tmp->lsword))
                        != 0 )
                       ls_tmp = ls_tmp->lnd_link;
                     else
                    {
                       intr_tmp = ls_tmp->intr_ptr;
                       break;
                    }
                 }
              }
           }
           else
           {
              intr_tmp = lb_tmp->intr_p;
              break;
           }
           if ( intr_tmp != MAXLINE )
              break;
           else
              lb_tmp = lb_tmp->up_lev;
        }
        return;
}

/*=========================================================*/
/*     This procedure is involked when a goto statement    */
```

```
/*     is reached.  It checks the exclusive extent         */
/*     of the same block.                                  */
/*     lb_tmp is point to the current bloick.              */
/*                                                         */
/*=========================================================*/


ckexclus(p)
struct pnode *p  ;
{


        while ( lb_tmp != p->cur_do )
        {
           if ((lb_tmp->delt & 000077 ) == do_lev + 1 )
           {
              if ((cond =strcmp(labword,lb_tmp->lword )) == 0)
              {
                 intr_tmp = lb_tmp->intr_p;
                 break;
              }
           }
           lb_tmp = lb_tmp->up_lev;
        }
        return;
}


fdlab (p)
struct pnode *p;
{

        if ( p != NULL )
        {
          lb_tmp = p->cur_do;
          chencls();
          if ( intr_tmp == MAXLINE )
          {  lb_tmp = p->doptr ;
             ckexclus(p);
          }
          if ( intr_tmp == MAXLINE)
          {
             p = p->ret_addr;
             fdlab (p);
          }
        }
        return ;
}
```

```
/*============================================================*/
/*                  Checks a character string                 */
/*============================================================*/
ckquote()
{
        if (lexst==25)
           strflag = ON ;
        return ;
}

/*============================================================*/
/*                  Checks a procedure name                   */
/*============================================================*/
chkproc()
{
struct pnode *pm;
int c;

        fdflag = NO;
        pm = cur_proc ->child;
        if ( pm != NULL )
        while ( pm != NULL )
        {  if (( c = strcmp(labword,pm->pword)) != 0)
               pm = pm ->subling;
           else
           {   fdflag = YES;
               yypref = pm;
               yysref = pm->pword;
               break;
           }
        }
        if ( !fdflag )
        {
           if ( cur_proc  != mod_name)
           {  pm = cur_proc->ret_addr;
              pm = pm->child;
              while ( pm != cur_proc )
              {  if (( c=strcmp(labword,pm->pword)) != 0)
                     pm = pm->subling ;
                 else
                 {   fdflag = YES;
                     yypref = pm;
                     yysref = pm->pword;
                     break;
                 }
              }
           }
        }
        return;
}
```

```
/*=========================================================*/
/*            Checks all the variables already declared    */
/*                     in the program.                     */
/*=========================================================*/
ckallnode()
{
auto struct tnode *t ;

auto struct tnode *t1;
        addflag = NO ;
        fdflag  = NO;
        p_tmp = cur_proc ;
        while (p_tmp->ret_addr != NULL || p_tmp != mod_name)
        {   p_tmp = p_tmp->ret_addr;
            t = p_tmp->locvar;
            if ( t != NULL )
            {  t1 = yybintree(t,labword);
               if ( fdflag )
               {
                  yytref = t1 ;
                  return ;
               }
             }
        }
        return ;
}

/*=========================================================*/
/*            Checks the variable whether is is delcared.  */
/*=========================================================*/
ckdecl(t)
int t;
{
struct dnode *yydalloc();

        if ( yytref->dnd_ptr == NULL )
        {   deltnode();
            ckallnode();
            addflag = YES;
            if ( ! fdflag )
             {    errout ( 2,yytref->tword);
                yytref = yybintree(cur_proc->locvar,labword);
                d_tmp = yydalloc(yytref);
                if ( t )
                   d_tmp->dtype = 0100;
                else
                   d_tmp->dtype = 0000;
             }
        }
        return;
}
```

```
/*=========================================================*/
/*                     Checks the parameter                */
/*=========================================================*/
chkprm()
{



        yypref = pars_p[0].pref;
        yytref = pars_p[2].tref;
        yysref = pars_p[2].sref;
        if ( yypref == NULL ) return;
        if ( calflag )
        {
           if ( yytref ->dnd_ptr == NULL )
          {   deltnode();
              ckallnode();
              addflag = YES;
              if ( ! fdflag )
                errout ( 5, yytref->tword);
              else
                prmpush (1) ;
           }
           else
              prmpush (1);
        }
        return;
}

/*=========================================================*/
/*                  Checks the variable type               */
/*=========================================================*/
ckvart(t)
int t;
{
        switch( var_type & 0007 )
        {
            case 1:  /* AT attribute */
                  {  if ( t )
                        mem_alc =- d_tmp->dim ;
                     else
                        mem_alc -- ;
                     d_tmp->mem_ptr = pars_p[2].mref;
                     break;
                  }
            case 2:  /* BASE variable */
                  {
                     if ( t )
                        mem_alc =- d_tmp->dim ;
```

```
                    else
                       mem_alc --;
                    d_tmp->mem_ptr = pars_p[2].mref;
                    d_tmp->mem_t = 2 ;
                    break;
                  }
        }
        yymref = d_tmp->mem_ptr ;
        yym_t = var_type;
        return;
}
/*=========================================================*/
/*       Counts the length of a giving string.             */
/*                    giving string.                       */
/*=========================================================*/


strlen(s)
char *s;
{
char *p;
        p = s;
        while ( *p != '\0' )
            p++ ;
        return ( p-s+1) ;
}


/*=========================================================*/
/*    Searches a giving token from the keyword table and   */
/*    returns a value which represents the position on     */
/*    the table or not found.                              */
/*=========================================================*/


yybinary (s)
char *s ;
{
char  *keytab [28] ;
int low, high, mid ;
        keytab [0] = "ADDRESS" ;
        keytab [1] = "AND";
        keytab [2] =  "AT" ;
        keytab [3] = "BASED";
        keytab [4] = "BY" ;
        keytab [5] = "BYTE";
        keytab [6] = "CALL";
        keytab [7] = "CASE";
        keytab [8] = "DECLARE";
        keytab [9] = "DO";
        keytab [10] = "ELSE";
```

```
        keytab [11] = "END";
        keytab [12] = "EOF" ;
        keytab [13] = "GO";
        keytab [14] = "GOTO";
        keytab [15] = "IF";
        keytab [16] = "INITIAL";
        keytab [17] = "INPUT";
        keytab [18] = "MOD";
        keytab [19] = "NOT" ;
        keytab [20] = "OR";
        keytab [21] = "OUTPUT";
        keytab [22] = "PROCEDURE" ;
        keytab [23] = "RETURN" ;
        keytab [24] = "STRING" ;
        keytab [25] = "THEN";
        keytab [26] = "TO" ;
        keytab [27] = "WHILE";
        low = 0;
        high = 27;
        while ( low <= high )
        {    mid = low + (high - low ) /2 ;
             if (( cond = strcmp (s,keytab[mid])) < 0)
                high = mid -1 ;
             else if ( cond > 0 )
                low = mid + 1 ;
                else return ( mid );
        }
        return (-1) ;
}
/*=========================================================*/
/*   This procedure is used to call binary search routine  */
/*   and return keyword.                                   */
/*=========================================================*/
srchkey ( s)
char *s ;
{
int index ;
        if (( index = yybinary (s) ) == -1 )
            return ( index ) ;
        else {
             switch (index)
             {
                case 0: return (ADDRESS) ;
                case  1: return (AND);
                case  2: return (AT);
                case  3: return (BASED) ;
                case  4: return (BY);
                case  5: return (BYTE);
                case  6:
                        {  calflag = ON ;
                           return (CALL);
```

```
        }
case  7:
        {
          dclflag = OFF ;
          lb_tmp = cur_proc ->cur_do ;
          if ( lb_tmp == NULL )
             printf ("error-- missing DO \n");
          else
          {  lb_tmp->dotype = 001000;
             casflag = ON ;
          }
          return (CASE);
        }
case  8:
        {
            if ( dclflag )
            dstflag = ON;
           return (DECLARE);
        }
case 9: /* allocate a doblock node for it */
         /* assume it is a simple do block */
        {
        labword = NULL;
        lb_tmp = cur_proc->cur_do;
        if (labflag)
        {  labflag = OFF ;
           if ( lb_tmp != NULL )
           {   ls_tmp = lb_tmp->lsta;
               lb_tmp->lsta = ls_tmp->lnd_link;
           }
           else
           {   ls_tmp = cur_proc->lsptr;
                cur_proc->lsptr = ls_tmp->lnd_link;
           }
           labword = ls_tmp->lsword;
           l_alc --;
        }
        do_type = 0; /* assume it is a simple do block */
        dclflag = ON ;
        lb = yylballoc();
        lb->up_lev = cur_proc->doptr ;
        cur_proc->doptr = lb;
        lb->lword = labword ;
        cur_proc ->cur_do = lb;
        return (DO);
        }
case 10: return (ELSE);
case 11: return (END);
case 12: return (EOF) ;
case 13: return (GO);
case 14: return (GOTO);
```

```
                case 15: return (IF);
                case 16: return (INITIAL);
                case 17: return (INPUT);
                case 18: return (MOD);
                case 19: return ( NOT);
                case 20: return (OR);
                case 21: return (OUTPUT);
                case 22:
                        {  pgmflag = ON ;
                           return (PROCEDURE);
                        }
                case 23:
                         return (RETURN);
                case 24: return (STRING);
                case 25: return (THEN);
                case 26: return (TO);
                case 27: /* set while flag on */
                        {  whlflag = ON ;
                           dclflag = OFF ;
                           lb_tmp = cur_proc->cur_do ;
                           lb_tmp->dotype = 003000;
                           return ( WHILE);
                        }
                }
                }
}

/*=========================================================*/
/*       Saves a character string in the text table.       */
/*=========================================================*/

char *strsave(sym)
char *sym;
{
char *start_p;
char *s;
char *t;
char *p;
int n ;


        start_p = textbuf ;
        while ( start_p < text_ptr )
        {    s = start_p ;
             t = sym ;
             for (; *s == *t; s++, t++ )
             {    if (*s == '\0' )
                       return (start_p);
             }
             while ( *s++)
                ;
```

```
            start_p = s ;
        }
talloc: n = strlen(sym);
        if (( text_ptr +n ) < (textbuf + TEXTLEN ) )
        {    p = text_ptr ;
             text_ptr =+ n ;
             strcpy ( p,sym );
             return (p);
        }
        else return (NULL);
}
/*=========================================================*/
/*               Compares two strings                      */
/*=========================================================*/

strcmp (s,t)
char *s, *t;
{
        for (; *s == *t; s++,t++)
            if ( *s == '\0' )
                return (0);
            return ( *s - *t);
}
/*=========================================================*/
/*             Copies one string into the text table       */
/*=========================================================*/
strcpy(s,t)
char *s, *t ;
{
        while ( *s++ = *t++ )
            ;
        *s = *t ;
}
/*=========================================================*/
/*        Converts a character to a decimal number         */
/*=========================================================*/

atoi (s)
char *s  ;
{
int n ;
        n = 0 ;
        for (; *s != '\0'; s++)
             n = 10 * n + *s - '0' ;
        return ( n );
}
/*=========================================================*/
/*   This procedure is used as index function.             */
/*=========================================================*/
```

```
yyindex (s,c)
char *s;
int  c ;
{
int i;

        for ( i=0; s[i] != '\0'; i++ )
            if ( s[i] == c )
                return ( i) ;
        return (-1) ;
}

/*=========================================================*/
/*        Allocates and links a new label statement        */
/*=========================================================*/
lslink()
{
        ls = yylsalloc();
        ls->lsword = labword ;
        lb_tmp = cur_proc ->cur_do;
        ls_tmp = lb_tmp->lsta;
        lb_tmp->lsta = ls ;
        ls->lnd_link = ls_tmp ;
        ls_tmp = lb_tmp->gosta;
}


*
```

```
#include  "m1.c"
#include  "def.c"
#define  yyclearin  yychar = -1
#define  yyerrok  yyerrflag = 0
extern  int yylval ;
extern int yyval;

/*************************************************************************
 **     This file comprises some routines which support the          **
 **     semantic action.                                             **
 *************************************************************************/

/*=========================================================*/
/*   Generates quadruple code for different arithmetic     */
/*   operations.                                           */
/*   The input value cd represents the different inter-    */
/*   mediate operation code.                               */
/*=========================================================*/

compush(cd)
int cd;
{
auto int i;

        i = tmpalloc();
        yyitalloc(cd,yymref,yym_t,pars_p[3].mref,pars_p[3].mref_t,i,NULL);
        yymref = temp_alc;
        yym_t = 0;
        yytref = NULL;
        return;
}

/*=========================================================*/
/*   Generates quadruple code for a restricted expression.*/
/*   The formal parameter "t" represents an intermediate   */
/*   operation code which is either  addition or subtra-   */
/*   ction.                                                */
/*=========================================================*/

locpush(t)
int t;
{
        tmpalloc();
        yyitalloc(t,yymref,1,pars_p[3].v,1,temp_alc,NULL);
        yymref = temp_alc;
        yym_t = 0;
        return;
}
```

```
/*=========================================================*/
/*   Generates a quadruple statement for a return statement.*/
/*=========================================================*/

retpush (t)
int t;
{

        if ( t )
        yyitalloc(24,NULL,NULL,NULL,NULL,NULL,NULL);
        else
        yyitalloc(25,yymref,yym_t,0,0,0,0);
        return ;
}

/*=========================================================*/
/*   Generates a quadruple statement for a call statement.*/
/*=========================================================*/

calpush()
{
                if ( yypref == NULL )
                   return;
                tmpalloc();
                yyitalloc(22,yymref,0,0,0,temp_alc,0);
                yymref = temp_alc;
                yym_t  = 0;
                return;
}

/*=========================================================*/
/*   Processes parameter passing in case of the actual     */
/*   parameter is a location reference or a constant.      */
/*=========================================================*/

prsprm(t)
char t;
{
        yypref = pars_p[0].pref;
        yymref = pars_p[2].mref;
        yysref = pars_p[2].sref;
        yym_t =pars_p[2].mref_t;
        if ( yypref != NULL )
        {
           if ( !calflag)
              errout ( 6, yysref );
           else
              prmpush(t);
        }
        return;
}
```

```
/*=========================================================*/
/*   Generates a quadruple statement for an actual         */
/*   parameter passing.                                    */
/*=========================================================*/

prmpush (t)
int t;
{
auto  struct tnode *form_root ;
auto  struct dnode *dc ;
auto  struct dnode *dp ;
auto  int ij;

        if ( t == 1 )
        dc = yytref ->dnd_ptr ;
        form_root = yypref->form_var;
        ij = 1;
        parmcnt ++ ;
        while ( ij < parmcnt && form_root != NULL )
        {  form_root = form_root ->left ;
           ij ++ ;
        }
        if ( form_root != NULL )
        {  dp = form_root ->dnd_ptr;
           chkpty(dp,dc,t);
           switch (t)
           {
               case 1:
             yyitalloc(23,dp->mem_ptr,dp->dtype,dc->mem_ptr,dc->dtype,
                       NULL,NULL);
                break;
             case 0:
             case 2:
             yyitalloc(23,dp->mem_ptr,dp->dtype,pars_p[2].mref,
                       pars_p[2].mref_t,NULL,NULL);
        }
        }
        else
           errout ( 6, yysref );
        return ;
}

/*=========================================================*/
/*   Check the type of actual and formal parameter.        */
/*=========================================================*/

chkpty(p,c,t)
struct dnode *p, *c;
char t;
```

```
{

        switch(t)
        {
                case 0:  /*  locator */
                        {
                          if ( p->dtype  >> 6  != 0)
                             errout ( 6, yysref);
                          break;
                        }
                case 1:  /*  variable */
                        {
                          if ( p->dtype  != c->dtype )
                             errout(6,yysref);
                          break;
                        }
                case 2:  /*  constant  */
                        {
                        printf ("dtype=%d, yymt=%d\n",p->dtype,yym_t);
                          if ( p->dtype  >> 6  != yym_t >> 6)
                             errout ( 6,yysref);
                          break;
                        }
        }
        return ;
}

prserr()
{
        yyerrok;
        yyclearin;
        return;

}


/*=========================================================*/
/*                  Checks the constant type               */
/*=========================================================*/

chconst()
{

        if ( yyval >= 0  && yyval  <= 255 )
           yym_t = 0100;
        else
           yym_t = 0;
        return ;
}
offcpfg()
{
```

```
        calflag = OFF;
        prmflag = OFF;
        return;

}

int popintr()
{
auto int m;
        m = intrstk[istkptr] ;
        istkptr --;
        return (m);
}


pushintr()
{

        istkptr ++;
        intrstk [istkptr] = intralc ;
        return ;
}

 ckconst (val)
int val;
{
int  ptr ;

        ptr = 0;
        while ( ptr < const_alc )
        {    if ( membuf [ ptr] == val )
                return (ptr);
                else ptr++ ;
        }

        membuf [ const_alc ] = val ;
        return ( const_alc ++ );
}

/*============================================================*/
/*                  Closes any do block                       */
/*============================================================*/

closedo()
{
auto int i;
auto struct dnode *d;
auto struct dclnode *dcl ;
auto struct tnode   *t;
```

```
do_type = lb_tmp->dotype ;
switch( do_type >> 9)
{
        case 0:  /* simple do block */
                {
                doflag = OFF ;
                if ( lb_tmp ->up_lev == NULL ) break ;
                dcl = lb_tmp ->dcl_link ;
                while ( dcl != NULL )
                {  t = dcl-> tptr ;
                   d = t->dnd_ptr ;
                   t ->dnd_ptr = d->dlink ;
                   dcl = dcl->next ;
                }
                break;
                }
        case 1:  /*  case do block */
                {
                  casflag = OFF ;
                  intralc =- 2;
                  istkptr -- ;
                  for ( i= 0; i < cascnt ; i++ )
                  {  m_tmp = popintr ();
                     intrtab [m_tmp].result = intralc + 1;
                  }
                  cascnt = 0;
                  casexp   = 0;
                  exp_t = 0;
                  break;
                }
        case 2:  /* do iterate block */
                { m_tmp = do_type & 000777;  /* step mem_ptr */
                  yyitalloc(1,lb_tmp->inx_mem,NULL,m_tmp,
                            lb_tmp->mt,lb_tmp->inx_mem,NULL);
                }
        case 3: /* for do iterate and do while block */
                {
                  m_tmp = popintr ();
                  intrtab [m_tmp].result = intralc + 2;
                  m_tmp = popintr();
                  yyitalloc(13,NULL,NULL,NULL,NULL,m_tmp,NULL);
                  break;
                }
}
lb_tmp->delt = 000100 | lb_tmp->delt ;
while ( lb_tmp->up_lev != NULL )
{
   lb_tmp = lb_tmp->up_lev;
   if ( lb_tmp->delt >> 6  == NO )
   {
      cur_proc->cur_do = lb_tmp ;
```

```
                do_lev = lb_tmp->delt & 000077 ;
                  return;
              }
          }
          do_lev = 0;
          return ;
}
/*=========================================================*/
/*   Checks other goto statement in the other block.       */
/*=========================================================*/

ckotgo(c,p)
char c;
struct pnode *p ;
{
auto struct donode *do_tmp;


do_tmp = p->doptr;
          while ( do_tmp != p->cur_do )
          {
             ls_tmp = do_tmp->gosta ;
             ckgosta(c);
             do_tmp = do_tmp->up_lev;
          }
          return;

}
/*=========================================================*/
/*   Checks all the goto statements in the program.        */
/*=========================================================*/

ckgoall (p)
struct pnode *p ;
{
auto struct donode *do_tmp;

          if ( p != NULL )
          {
            do_tmp= p->cur_do ;
            ls_tmp = do_tmp->gosta ;
            ckgosta(1);
            ckotgo ( 1,p);
            if ( lb_tmp->up_lev == 0 && cur_proc == mod_name )
            {  ckgoall ( p->child );
               ckgoall ( p->subling );
            }
          }
          return ;
}
```

```
/*=========================================================*/
/*   Checks all goto statement in the same block.          */
/*=========================================================*/

ckgosta (t)
char  t;
{

auto  int  intr_t;
auto  struct lnode *tmp1, *tmp2 ;

        /* ls_tmp points to a uncompleted goto statement */
        /* ls points to the new label statement          */
        while ( ls_tmp != NULL )
        {  if (( cond =strcmp(labword,ls_tmp->lsword)) == 0)
                /* recover this previous uncompleted goto statement */
           {   intr_t = ls_tmp->intr_ptr;
               intrtab [intr_t].result = ls->intr_ptr;

               tmp1 = ls_tmp->bck_link ;
               tmp2 = ls_tmp->lnd_link;
               if ( tmp1 != NULL )
                 tmp1->lnd_link = tmp2;
               else
                {  switch (t)
                   {    case  1:  /* go to statement is in a do block */
                                  lb_tmp->gosta = tmp2;
                                  break;
                        case  2:  /* go to statement is not in a do block */
                                  {  cur_proc->gsptr = tmp2;
                                     break;
                                  }
                    }
                }
               if ( tmp2 != NULL )
                  tmp2->bck_link = tmp1 ;

             }
             ls_tmp = ls_tmp->lnd_link ;
           }
           return ;
        }

/*=========================================================*/
/*                Checks the variable type                 */
/*=========================================================*/

ckaddr ()
{
        /* check variable is it an address type */
        if ( rsflag == ON )
```

```
        {  d_tmp = yytref->dnd_ptr ;
           t_type = d_tmp->dtype;
           switch ( t_type >> 6)
           {  case 0:
                    break;

              default:
                    errout ( 7, yytref->tword);
                    break;
           }
        }
        return ;
}

/*============================================================*/
/*   Thie procedure is used to allocate declaration node.   */
/*============================================================*/
struct dclnode *yydclalloc()
{

        if ( dcl_alc < dcltab + DCLSIZE )
        {  dcl_alc ->tptr = NULL ;
           dcl_alc ->next = NULL ;
           return ( dcl_alc ++ );
        }
        else
        {
           errout(9,NULL );
           return ( NULL );
        }
}

/*============================================================*/
/*                    Allocate tnode                          */
/*============================================================*/


struct tnode *yytalloc()
{
char *strsave() ;
struct tnode *p;

        if (tnd_alc < tndtab + TNDSIZE )
        {
                tnd_alc -> tword = strsave (idbuf) ;
                tnd_alc -> left  = NULL ;
                tnd_alc -> lf_back = NULL ;
                tnd_alc -> rg_back = NULL ;
                tnd_alc -> right = NULL ;
                tnd_alc -> dnd_ptr = NULL ;
                return (tnd_alc++) ;
```

```
        }
        else
        {
           errout ( 9,NULL );
           return (NULL );
        }
}

/*=========================================================*/
/*                Searches the formal parameter            */
/*=========================================================*/
struct tnode *srchform(ptr,sym)
struct tnode *ptr;
char   *sym;
{

        if (ptr != NULL)
        {  while ( (cond = strcmp(sym,ptr->tword)) != 0 )
           {  if (ptr->left == NULL )
                  return (NULL);
              else ptr = ptr->left ;
           }
          return (ptr);
        }
        else return (NULL);

}

/*=========================================================*/
/*    Deletes the unnecessary node in the symbol table.    */
/*=========================================================*/
deltnode()
{

auto struct tnode *tmp;

        if (yytref == cur_proc->locvar )
            cur_proc->locvar = NULL ;
        else {
        if ( (tmp= yytref->lf_back) != NULL )
             tmp->left = NULL;
        if ( (tmp = yytref->rg_back) !=NULL)
             tmp->right = NULL ;
        }
        return;
}

/*=========================================================*/
/*                    Allocates pnode                      */
/*=========================================================*/
struct pnode *yypalloc ( )
```

```
{
char *strsave() ;
struct pnode *p;

        if (pnd_alc < pndtab + PNDSIZE )
        {
                pnd_alc-> ret_addr = NULL;
                pnd_alc -> locvar   = NULL ;
                pnd_alc -> form_var = NULL;
                pnd_alc -> intrpnt = MAXLINE ;
                pnd_alc -> doptr   = NULL ;
                pnd_alc -> cur_do = NULL;
                pnd_alc -> lsptr    = NULL;
                pnd_alc -> gsptr    = NULL;
                pnd_alc -> child    = NULL ;
                pnd_alc -> subling  = NULL ;
                pnd_alc -> ptype    = NULL ;
                pnd_alc -> pword    = strsave(idbuf) ;
                do_lev = 0;
                return (pnd_alc++) ;
        }
        else
        {
           errout ( 9, NULL );
           return (NULL );
        }
}


/*=============================================================*/
/*                    Allocates dnode                          */
/*=============================================================*/

struct dnode  *yydalloc (t_ptr)
struct tnode  *t_ptr ;
{
auto struct donode *d;
auto struct dclnode *dcl;


        if (dnd_alc < dndtab + DNDSIZE )
        {
                dcl = yydclalloc();
                dcl ->tptr = t_ptr;
                d = cur_proc ->cur_do ;
                dcl->next = d->dcl_link;
              , d->dcl_link = dcl ;
                dnd_alc -> dlink = t_ptr -> dnd_ptr ;
                dnd_alc ->s_ptr = NULL;
                t_ptr -> dnd_ptr = dnd_alc ;
                dnd_alc -> dtype = NULL;
```

```
                dnd_alc -> dim = NULL;
                dnd_alc -> mem_t = NULL ;
                if (mem_alc <  MEMSIZE )
                {   dnd_alc ->mem_ptr = mem_alc ;
                    mem_alc ++ ;
                }
                else
                {
                   errout (10,NULL );
                   dnd_alc ->mem_ptr = 0;
                }
                return (dnd_alc ++);
        }
        else
        {
           errout (9,NULL );
           return ( NULL );
        }
}
/*=========================================================*/
/*               Allocates a label statement               */
/*=========================================================*/

struct  lnode *yylsalloc()
{
        if ( l_alc < lstab + LABSTSZ )
        {
            l_alc -> lsword = NULL;
            l_alc -> intr_ptr = intralc + 1 ;
            l_alc -> lnd_link = NULL;
            l_alc -> bck_link = NULL;
            return (l_alc ++ );
        }
        else
        {
           errout (9,NULL );
           return ( NULL );
        }
}


/*=========================================================*/
/*               Allocates a do block                      */
/*=========================================================*/

struct  donode  *yylballoc()
{
        if ( do_alc < dotab + DOBKSZ )
        {
           do_lev  ++ ;
            do_alc -> lword = NULL ;
```

```
            do_alc -> dcl_link = NULL ;
            do_alc -> lsta = NULL;
            do_alc -> gosta = NULL;
            do_alc -> up_lev = NULL;
           do_alc -> delt = do_lev ;
            do_alc -> intr_p = intralc + 1 ;
            do_alc -> dotype =  do_type ;
            do_alc -> inx_mem = NULL;
            do_alc -> mt        = NULL;
            return ( do_alc ++ );
         }
         else
         {
            errout ( 9, NULL );
            return ( NULL );
         }
}

/*============================================================*/
/*        Generates a specified quadruple statement.          */
/*============================================================*/

yyitalloc(opcd,or1,or1t,or2,or2t,res,rest)
int opcd,or1,or1t,or2,or2t,res,rest;
{

         intralc ++ ;
         if (intralc < MAXLINE )
         {  if (cur_proc ->intrpnt == MAXLINE )
                cur_proc ->intrpnt = intralc ;
            intrtab [intralc].opcode = opcd;
            intrtab [intralc].opr1   = or1;
            intrtab [intralc].opr1_t = or1t;
            intrtab [intralc].opr2   = or2;
            intrtab [intralc].opr2_t = or2t;
            intrtab [intralc].result = res;
            intrtab [intralc].result_t = rest;
            if ( whlflag == ON )
            {  whlflag = OFF ;
                 pushintr();
            }
            return (intralc);
         }
         else
         {
            errout ( 9, NULL );
            return ( NULL );
         }
}
%
```

```
#include  "m1.c"
#include  "def.c"
extern  int  yyval;
extern  int yylval ;
extern  fin, fout;
int op, or1, or2, or1t, or2t, res ;
/*****************************************************************
 **                                                             **
 **                *===*     EXECUTION PHASE     *===*         **
 **     This file contains a routine called "yyaccept" which    **
 **     executes the intermediate codes when the parser success- **
 **     fully parses the input stream.                          **
 **                                                             **
 *****************************************************************/
yyaccept()
{
        auto int i ;
        auto int indx;
        if ( !errflag)
        printf ("  no compile-time error \n");
        if ( yydebug)
        {
           printf ("  interpreter table \n  ");
           printf (" intralc = %3d \n",intralc);
           printf ("          opcode      opr1    opr1_t    opr2      opr2_t");
           printf ("    result    result_t\n");
           for (i =1; i <= intralc; i++ )
           {
           printf ("%5d     %5d     ",i,intrtab[i].opcode);
           printf ("%5d     %5d     ",intrtab[i].opr1,intrtab[i].opr1_t);
           printf ("%5d     %5d     ",intrtab[i].opr2,intrtab[i].opr2_t);
           printf ("%5d     %5d \n",intrtab[i].result,intrtab[i].result_t);
        }
        }
         rstkptr = &retstk[0] -1 ;
         indx = mod_name -> intrpnt ;
         while ( indx <= intralc  && !errflag )
         {
         op = intrtab [indx].opcode ;
         or1 = intrtab [indx].opr1;
         or1t = intrtab [indx].opr1_t ;
         or2 = intrtab [indx].opr2 ;
         or2t = intrtab [indx].opr2_t;
         res = intrtab [indx].result ;
                 switch ( op )
                 {
                 case  1:
                 case  2:
                 case  3:
                 case  4:
```

```
case  5:
case  6:
case  7:
case  8:
case  9:
case 10:
case 11:
case 12:{
        aricomp();
        indx ++ ;
        break;
        }
case 13:{
        if ( res == 0 )
        {
          printf (" illegal goto statement \n");
          errflag = ON ;
        }
        indx = res ;
        break;
        }
case 14:{
        chrtype ( &or2, &or2t);
        or1 = chltype ( or1,or1t);
        membuf [or1] = or2 ;
        rvltype ( or1, or1t);
        indx ++ ;
        break;
        }
case 15:{
        chrtype ( &or1, &or1t);
        if ( or1 )
           indx ++ ;
        else
           indx = res ;
        break;
        }
case 16:{
        aricomp();
        indx ++ ;
        break;
        }
case 20:{
        chrtype ( &or1, &or1t );
        membuf [res ] = ! or1 ;
        indx ++ ;
        break;
        }
case 21:{
        or1 = chltype (or1, or1t);
        membuf [or1] = - membuf [or1] ;
```

```
                rvltype ( or1, or1t);
                indx ++ ;
                break;
                }
        case 22:{
                rstkptr ++ ;
                *rstkptr = indx + 1;
                indx = or1 ;
                break;

                }
        case 23:{   /*  pass parameter */
                chrtype( &or2, &or2t );
                membuf [or1] = or2 ;

                indx ++ ;
                break;
                }
        case 24:{   /*  untyped return  */
                indx = *rstkptr ;
                rstkptr -- ;
                break;
                }
        case 25:{   /*  typed return  */
                indx = *rstkptr;
                rstkptr -- ;
                chrtype ( &or1, &or1t );
                i = intrtab [indx-1].result ;
                membuf [i] = or1 ;
                break;
                }
        case 26:{       /*  input a number */
                or1 = chltype (or1, or1t);
                membuf [ or1 ] = innum ();
                rvltype ( or1,or1t);
                indx ++;
                break;
                }
        case 27:{       /*  output a number */
                or1 = chltype(or1,or1t);
                rvltype(or1, or1t);
                printf (" in line %d output value=%d\n",res,
                        membuf[or1]);
                indx++ ;
                break;
                }
        }
}

if ( yydebug)
 {
```

```
        for ( i = 0; i < mem_alc ; i ++ )
        {
        printf (" memory  %d =  %d \n", i, membuf [i] );
        }
       }
        return;
}

/*************************************************************
 **    Input a number from user's terminal                  **
 *************************************************************/

innum()
{
char c;
        constptr = constbuf ;
        printf (" please input number--");
        cin = fin = 0;
        while ( ( c= getchar() ) != '\n' )
        {
          if ( constptr < constbuf + CONSTLEN -1 )
             *constptr ++ = c ;
        }
        *constptr = '\0' ;
        return ( atoi (constbuf )) ;
}

/*************************************************************
 **      Check the type of right-hand side value             **
 *************************************************************/
chrtype(i,it)
int  *i, *it;
{
        switch ( *it  & 0007)
        {
        case 0:
                *i = membuf [*i] ;
                break;
        case 1:
                break;
                case 2:
                *i = membuf [ membuf[*i]] ;
                        break;
                case 3:
                        *i = membuf [*i] ;
                        break;
        }
        switch ( *it >> 6)
        {
        case 0:
                break;
```

```
        case 1:
                *i = *i &000377 ;
        }
        return;
}

/*************************************************************
 **        check the type of left-hand side value          **
 *************************************************************/
chltype (or, ort)
int or, ort;
{

        switch ( ort & 0007 )
        {
          case  2:
                  or = membuf [ or ];
        }
        return ( or);

}

/*************************************************************
 **    Revise the type of left-hand side value             **
 *************************************************************/
rvltype ( or,ort)
int or, ort;
{
        switch ( ort >> 6 )
        {
          case  1:
                  membuf [ or ] = membuf [ or] & 000377 ;
        }

}

aricomp()
{
        chrtype ( &or1, &or1t);
        chrtype ( &or2, &or2t);
        switch ( op )
        {
                case 1:   /*  addition */
                        membuf [res] = or1 + or2 ;
                        break;
                case 2:    /*  subtraction  */
                        membuf [res] = or1 - or2 ;
                        break;
                case 3:   /*  multiplication  */
                        membuf [res] = or1  * or2 ;
                        break;
```

```
                case 4:   /*  division       */
                        if ( or2 == 0 )
                        {
                          printf (" division by 0 \n");
                          errflag = ON ;
                        }
                        membuf [res] = or1 / or2 ;
                        break;
                case 5:   /*  logical OR    */
                        membuf [res] = or1  ||  or2 ;
                        break;
                case 6:   /* logical AND   */
                        membuf [res] = or1  &&  or2 ;
                        break ;
                case 7:   /* logical less then  */
                        membuf [res] = or1  < or2 ;
                        break ;
                case 8:   /* logical  greater than  */
                        membuf [res] = or1 >  or2 ;
                        break;
                case 9:   /*   logical  equal to  */
                        membuf [res] = or1 == or2 ;
                        break;
                case 10:   /*  logical   less than or equal to  */
                        membuf [res] = or1 <= or2 ;
                        break;
                case 11:   /*  logical greater than or equal to  */
                        membuf [res] = or1 >= or2 ;
                        break;
                case 12:   /*  logical not equal to  */
                        membuf [res] = or1 != or2 ;
                        break;
                case  16:   /*  MOD   */
                        membuf [res] = or1  %  or2 ;
        }
        return ;
}

%
```

```
#
# define ADDRESS 257
# define AND 258
# define AT 259
# define BASED 260
# define BY 261
# define BYTE 262
# define CALL 263
# define CASE 264
# define DECLARE 265
# define DATA 266
# define DO 267
# define ELSE 268
# define END 269
# define EOF 270
# define GO 271
# define GOTO 272
# define ID 273
# define IF 274
# define INITIAL 275
# define INPUT 276
# define MOD 277
# define NUMBER 278
# define OR 279
# define OUTPUT 280
# define PROCEDURE 281
# define RETURN 282
# define STRING 283
# define THEN 284
# define TO 285
# define WHILE 286
# define NOT 287
# define UMINUS 288
#include  "m1.c"
#include  "def.c"
#define yyclearin yychar = -1
#define yyerrok yyerrflag = 0
extern int yychar, yyerrflag;

int yyval 0;
int *yypv;
int yylval 0;
yyactr(__np__){

switch(__np__){

case 4:
                        {
                        mod_name = yypref;     } break;
case 5:
                        {
```

```
                    if (mod_name != NULL )
                    {  if (yytref->dnd_ptr == NULL )
                        deltnode();
                       else
                        errout ( 1, yytref->tword);
                       labflag = ON ;
                       labword = yytref ->tword;
                       lslink();
                       p_tmp = cur_proc ;
                       ckgoall (p_tmp);
                    }   } break;
case 6:
                     {
                     dclflag = OFF;
                     dstflag = OFF;    } break;
case 13:
                    {
                    prserr();            } break;
case 14:
                    {
                    clrtmp();
                    chkcase();           } break;
case 15:
                    {
                    clrtmp();
                    chkcase();        } break;
case 22:
                    {
                    prserr();
                    printf (" probably missing ';'\n"); } break;
case 26:
                    {
                    prserr();
                    printf(" probably missing ';'\n");  } break;
case 28:
                    {
                    prserr();
                    printf(" probably missing ';'\n");  } break;
case 31:
                    {
                    prserr();   } break;
case 32:
                    {
                    m_tmp = popintr();
                    intrtab [m_tmp].result = intralc + 1;  } break;
case 33:
                    {
                    m_tmp = popintr ();
                    intrtab [m_tmp].result = intralc + 1;  } break;
case 34:
                    {
```

```
                yyitalloc ( 15,pars_p[2].mref,pars_p[2].mref_t,NULL,
                        NULL,NULL,NULL);
                pushintr();       } break;
case 35:
                {
                prserr();
                printf (" probably missing  keyword THEN \n");  } break;
case 36:
                {
                m_tmp = intrstk[istkptr] ;
                intrtab [m_tmp].result = intralc + 2 ;
                yyitalloc(13,NULL,NULL,NULL,NULL,NULL,NULL);
                intrstk[istkptr] = intralc ;   } break;
case 40:
                {
                staflag = ON ;   } break;
case 44:
                {
                lb_tmp = cur_proc->cur_do ;
                lb_tmp->dotype = lb_tmp->dotype | 02000 ;
                lb_tmp->inx_mem = yymref ;
                yyitalloc(14,yymref,NULL,pars_p[3].mref,pars_p[3].mref_t,
                        NULL,NULL);
                intr_tmp = tmpalloc();
                yyitalloc(10,yymref,NULL,pars_p[4].mref,pars_p[4].mref_t,
                        intr_tmp,NULL);
                pushintr();
                yyitalloc(15,temp_alc,NULL,NULL,NULL,NULL,NULL);
                pushintr();
                staflag = ON ;
                dclflag = OFF ;    } break;
case 45:
                {
                lb_tmp = cur_proc->cur_do ;
                do_type = lb_tmp->dotype;
                m_tmp = ckconst (1);
                do_type = do_type | m_tmp ;
                lb_tmp->dotype = do_type;
                yymref = pars_p[2].mref;
                yym_t  = pars_p[2].mref_t;  } break;
case 46:
                {
                prserr();
                printf("    probably missing ';'\n");  } break;
case 47:
                {
                prserr();
                printf(" probably missing ';'\n");  } break;
case 48:
                {
                lb_tmp = cur_proc->cur_do ;
```

```
                    do_type = lb_tmp->dotype ;
                    if (pars_p[4].mref_t == 2)
                       lb_tmp->mt = 2;
                    do_type = do_type | pars_p[4].mref;
                    lb_tmp->dotype = do_type;
                    yymref = pars_p[2].mref;
                    yym_t = pars_p[2].mref_t ;  } break;
case 49:
                    {
                    if  (casflag ) doflag = ON ;
                      dclflag = ON;  } break;
case 50:
                    {
                    yyitalloc(15,pars_p[2].mref,pars_p[2].mref_t,NULL,NULL,
                           NULL,NULL);
                    pushintr();   } break;
case 51:
                    {
                    casexp = pars_p [2].mref ;
                    exp_t = pars_p[2].mref_t ;
                    cascnt = 0;
                    caselector ();   } break;
case 57:
                    {
                    yypref = pars_p [2].pref ;
                    yysref = yypref->pword;
                    pgmflag = OFF ;
                    prmflag = ON;
                    if ( cur_proc->child != NULL)
                    {
                      ptmp = cur_proc ->child;
                      while ( ptmp->subling != NULL)
                         ptmp = ptmp->subling ;
                      ptmp->subling = yypref ;
                    }
                    else
                       cur_proc->child = yypref ;
                    yypref->ret_addr = cur_proc ;
                    cur_proc = yypref ;
                    yytref = NULL;   } break;
case 62:
                    {
                    chkprm ();
                    offcpfg();  } break;
case 63:
                    {
                    prsprm(0);
                    offcpfg();   } break;
case 64:
                    {
                    prsprm(1);
```

```
                    offcpfg();   } break;
case 66:
                    {
                    chkprm();  } break;
case 67:
                    {
                    prsprm(0);  } break;
case 68:
                    {
                    prsprm(1);  } break;
case 69:
                    {
                    lb_tmp = cur_proc->cur_do ;
                    labword = NULL;
                    if ( lb_tmp->delt  >> 6 == 1 )
                    {
                       cur_proc = cur_proc->ret_addr ;
                       if (cur_proc == NULL )
                         errout ( 3, NULL );
                    }
                    else
                        closedo ();  } break;
case 70:
                    {
                    yytref = pars_p[2].tref;
                    if (yytref->dnd_ptr == NULL )
                    {  deltnode();
                       labword = yytref->tword; }
                    else
                    {     errout ( 1, yytref->tword );
                          labword = NULL ; }
                    lb_tmp = cur_proc ->cur_do ;
                    if ( lb_tmp->delt >> 6 == 1 )
                    {  if ( cur_proc->ret_addr == NULL)
                          errout ( 3, NULL );
                       else
                       {  if ( labword != NULL )
                          { if (( cond=strcmp(labword,cur_proc->pword))
                                != 0 )
                              errout ( 11, labword );
                           }
                          cur_proc = cur_proc->ret_addr ;
                       }
                    }
                    else closedo();  } break;
case 74:
                    {
                    yymref = pars_p [2].mref ;
                    yym_t  = pars_p [2].mref_t ;
                    retpush (0) ; } break;
case 75:
```

```
                {
                retpush (1) ;  } break;
case 76:
                {
                calpush();  } break;
case 77:
                {
                calpush();  } break;
case 78:
                {
                yytref = pars_p[2].tref;
                labword = yytref-> tword;
                if ( !fdflag)
                {  deltnode();
                   chkproc ();
                   if ( !fdflag)
                     errout ( 4, labword);
                   else
                     yymref = yypref->intrpnt;
                }
                else
                errout ( 4, labword); } break;
case 79:
                {
                intr_tmp = MAXLINE;
                yytref = pars_p[2].tref;
                labword = yytref ->tword;
                if ( yytref ->dnd_ptr == NULL )
                deltnode ();
                else
                   errout ( 1,labword);
                p_tmp = cur_proc ;
                fdlab ( p_tmp);
                  if ( intr_tmp == MAXLINE)
                  {  lb_tmp = cur_proc ->cur_do;
                     ls_tmp = lb_tmp->gosta;
                     ls= lb_tmp->gosta = yylsalloc();
                     ls->lnd_link = ls_tmp;
                     if ( ls_tmp != NULL )
                          ls_tmp->bck_link = ls;
                     ls->lsword = labword ;
                     ls->intr_ptr = intralc;  }
                  else
                     intrtab [intralc].result = intr_tmp; } break;
case 80:
                  {
                  yyitalloc(13,NULL,NULL,NULL,NULL,NULL,NULL); } break;
case 81:
                  {
                  yyitalloc(13,NULL,NULL,NULL,NULL,NULL,NULL); } break;
case 82:
```

```
                    {
                    clrdecl ();  } break;
case 83:
                    {
                    clrdecl ();  } break;
case 85:
                    {
                    d_tmp = yydalloc(yytref);
                    d_tmp->dtype = var_type;
                    m_tmp = d_tmp->mem_ptr;
                    membuf [m_tmp] = pars_p[2].v ;  } break;
case 87:
                    {
                    var_type = var_type  |  0003; } break;
case 89:
                    {
                    d_tmp = yydalloc(yytref);
                    d_tmp->dtype = var_type;
                    ckvart(0);  } break;
case 90:
                    {
                    d_tmp = yytref->dnd_ptr ;
                    d_tmp->dtype = var_type;
                    ckvart (1);  } break;
case 91:
                    {
                    if ( yym_t & 0007 == 2 )
                    {
                     errout ( 11,yytref->tword);
                     break; }
                    else
                    {
                     for ( cond= 0; rstkptr >=retstk; cond++ )
                     {
                        membuf [yymref+cond] = retstk[cond] ;
                        rvltype( yymref+cond, yym_t);
                        rstkptr -- ;  }
                    }
                     rstkptr = &retstk[0] -1 ;  } break;
case 92:
                     {
                     d_tmp = yytref->dnd_ptr ;
                     d_tmp->dim = pars_p [2].v ;
                     mem_alc =+  pars_p [2].v -1 ;
                     yymref = d_tmp->mem_ptr; } break;
case 94:
                     {
                     d_tmp = yydalloc(yytref);
                     yymref = d_tmp->mem_ptr ; } break;
case 98:
                     {
```

```
                    var_type = var_type | 0001 ; } break;
case 100:
                    {
                    yymref = pars_p [2].mref; } break;
case 101:
                    {
                    var_type = var_type | 0100 ; } break;
case 102:
                    {
                    var_type = var_type | 0000; } break;
case 103:
                    {
                    var_type = var_type | 0002 ;
                    yytref = pars_p [2].tref;
                    labword = yytref->tword;
                    p_tmp = cur_proc;
                    ckdecl (0);
                    d_tmp = yytref->dnd_ptr ;
                    t_type = d_tmp->dtype;
                    if (( t_type >> 6 ) != 0  ||
                       (t_type & 0001 ) != 0 )
                    {
                       errout (7, labword);
                       d_tmp->dtype = 0000;
                    }
                    yymref = d_tmp->mem_ptr ; } break;
case 104:
                    {
                    if ( rstkptr < retstk + 19 )
                    {
                       rstkptr ++ ;
                       *rstkptr = pars_p[2].v ;
                    }  } break;
case 105:
                    {
                    rstkptr = &retstk[0] -1 ; } break;
case 106:
                    {
                    if ( rstkptr < retstk + 19 )
                    {
                       rstkptr ++;
                       *rstkptr = pars_p[2].v ; } } break;
case 107:
                    {
                    yyitalloc(14,yymref,yym_t,pars_p[3].mref,pars_p[3].mref_t,
                         NULL,NULL);
                    ckaddr (); } break;
case 108:
                    {
                    yyitalloc(14,yymref,yym_t,intrtab[intralc].opr2,
                         intrtab[intralc].opr2_t,NULL,NULL);
```

```
                    ckaddr ();  } break;
case 109:
                    {
                    yyitalloc(26,yymref,yym_t,0,0,0,0 );  } break;
case 110:
                    {
                    yymref = pars_p[3].mref ;
                    yym_t = pars_p[3].mref_t;
                    yyitalloc(27,yymref,yym_t,0,0,yyline,0); } break;
case 114:
                    {
                    rsflag = ON ;  } break;
case 115:
                    {
                    yytref = pars_p [2].tref ;
                    yysref = pars_p [2].sref;
                    labword = yytref->tword;
                    p_tmp = cur_proc;
                    ckdecl (1);
                    d_tmp = yytref->dnd_ptr;
                    yymref = d_tmp->mem_ptr ;
                    yym_t = 1;   } break;
case 116:
                    {
                    if (dstflag)
                       yymref = yymref + pars_p [3].v;
                    else
                       locpush (1) ;  } break;
case 117:
                    {
                    if (dstflag)
                         yymref = yymref + pars_p [3].v ;
                    else
                         locpush (2) ; } break;
case 120:
                    {
                    compush (5); } break;
case 122:
                    {
                    compush (6);  } break;
case 124:
                    {
                    intr_tmp = tmpalloc();
                    yyitalloc(20,pars_p[2].mref,pars_p[2].mref_t,NULL,NULL,
                         intr_tmp,NULL);
                    yymref = temp_alc;
                    yytref = NULL ;  } break;
case 126:
                    {
                    compush (pars_p[2].v) ; } break;
case 127:
```

```
                    {
                     yyval = 9; } break;
case 128:
                    {
                     yyval = 7 ;  } break;
case 129:
                    {
                     yyval = 8 ;  } break;
case 131:
                    {
                     yyval = 10; } break;
case 132:
                    {
                     yyval = 11; } break;
case 133:
                    {
                     yyval = 12 ;  } break;
case 135:
                     {
                     compush (1) ; } break;
case 136:
                     {
                     compush (2); } break;
case 138:
                     {
                     compush (3); } break;
case 139:
                     {
                     compush (4); } break;
case 140:
                     {
                     compush (16) ; } break;
case 144:
                     {
                     intr_tmp = tmpalloc();
                     yyitalloc(21,pars_p[2].mref,pars_p[2].mref_t,NULL,NULL,
                         intr_tmp,NULL); } break;
case 145:
                     {
                     yymref = pars_p [2].mref;
                     yym_t = pars_p [2].mref_t; } break;
case 147:
                     {
                     tmpalloc();
                     intr_tmp = yym_t  |  1;
                     yyitalloc(1,yymref,intr_tmp,pars_p[2].mref,pars_p[2].mref_t,
                         temp_alc,2);
                     yymref = temp_alc ;
                     yym_t = yym_t | 2 ;  } break;
case 148:
                     {
```

```
                        chconst();  } break;
case 151:
                        {
                        labword = yytref ->tword;
                        p_tmp = cur_proc ;
                        if ( prmflag ) break;
                           ckdecl (1);
                        d_tmp = yytref->dnd_ptr;
                        yymref = d_tmp->mem_ptr ;
                        yym_t = d_tmp->dtype ;  } break;
}
}
int yyerrval 256;

int yyact[] {0,4369,8197,0,4096,16384,0,4366,8198,12289
,4363,8202,0,12292,4154,8203,0,12290,12291,4359
,8233,4363,8237,4365,8207,4367,8235,4368,8236,4369
,8211,4370,8229,4376,8225,4378,8232,0,4359,8233
,4361,8244,4363,8237,4365,8207,4367,8235,4368,8236
,4369,8211,4370,8229,4376,8225,4377,8248,4378,8232
,0,4155,8249,0,12293,4352,8252,4359,8233,4363
,8237,4365,8207,4367,8235,4368,8236,4369,8211,4370
,8229,4376,8225,4378,8232,0,12296,12299,4369,8254
,4155,8253,0,4359,8233,4363,8237,4365,8207,4367
,8235,4368,8236,4369,8211,4370,8229,4376,8225,4378
,8232,0,12302,12303,4154,8203,12439,4352,8258,4155
,8257,0,12311,12312,4352,8260,4155,8259,0,4352
,8262,4155,8261,0,12317,4352,8264,4155,8263,0
,4359,8233,4363,8237,4367,8235,4368,8236,4369,8211
,4370,8229,4376,8225,4378,8232,0,4140,8270,4157
,8271,0,4369,8274,0,12360,12361,4155,8275,4136
,8278,0,4157,8271,0,12330,12331,4369,8280,0
,4369,8274,4374,8297,4379,8298,4383,8289,4141,8296
,4136,8299,4142,8287,0,4136,8300,12434,4369,8274
,4374,8297,4379,8298,4383,8289,4141,8296,4155,8303
,4136,8299,4142,8287,0,4369,8304,0,4359,8233
,4363,8237,4365,8207,4367,8235,4368,8236,4369,8211
,4370,8229,4376,8225,4378,8232,0,4381,8307,0
,12369,4360,8313,4369,8274,4382,8312,4155,8249,0
,4361,8244,4377,8248,12294,12297,4352,8252,4359,8233
,4363,8237,4365,8207,4367,8235,4368,8236,4369,8211
,4370,8229,4376,8225,4378,8232,0,12305,4155,8316
,4140,8317,0,12308,4369,8320,0,4359,8233,4361
,8244,4363,8237,4365,8207,4367,8235,4368,8236,4369
,8211,4370,8229,4376,8225,4377,8248,4378,8232,0
,4353,8330,4358,8329,4155,8326,4136,8278,0,4369
,8331,0,12344,12337,12295,12300,4155,8332,0,12357
,4155,8333,0,12304,12359,12309,4155,8334,0,12313
,4155,8335,0,12315,12316,12318,4155,8336,0,12320
,4359,8233,4363,8237,4367,8235,4368,8236,4369,8211
,4370,8229,4376,8225,4378,8232,0,4364,8338,12302
```

```
,4369,8274,4372,8340,4374,8297,4379,8298,4383,8289
,4141,8296,4136,8299,4142,8287,0,12400,12399,12396
,4140,8270,4157,8271,0,12439,12364,4155,8342,0
,4369,8274,4374,8297,4379,8298,4142,8287,0,12353
,4369,8274,0,12367,4352,8348,4380,8347,0,4375
,8349,12401,12402,4354,8350,12407,4139,8351,4141,8352
,12406,12409,4369,8353,0,12411,4369,8274,4374,8297
,4379,8298,4141,8296,4136,8299,0,4139,8356,4141
,8357,4385,8358,4156,8359,4158,8360,4386,8362,4387
,8363,4388,8364,12413,4373,8367,4138,8365,4143,8366
,12422,12425,12429,12430,12431,4369,8274,4374,8297,4379
,8298,4141,8296,4136,8299,0,12436,12437,12438,4137
,8370,0,4155,8371,0,12363,12366,12325,12329,12368
,12326,4155,8372,0,4155,8373,0,12306,12298,12307
,12370,4371,8380,12372,4353,8330,4356,8390,4358,8329
,4362,8387,4187,8383,0,4353,8330,4356,8390,4358
,8329,0,4374,8393,4138,8394,0,4359,8233,4361
,8244,4363,8237,4365,8207,4367,8235,4368,8236,4369
,8211,4370,8229,4376,8225,4377,8248,4378,8232,0
,4352,8252,4359,8233,4363,8237,4365,8207,4367,8235
,4368,8236,4369,8211,4370,8229,4376,8225,4378,8232
,0,12349,12340,4155,8398,0,4353,8330,4358,8329
,4155,8399,0,12389,12390,12345,12301,12358,12310,12314
,12319,12321,12324,12395,12397,12365,4137,8401,4140,8402
,0,4137,8403,4140,8404,0,4137,8405,4140,8406
,0,12398,12322,12323,4369,8274,4374,8297,4379,8298
,4383,8289,4141,8296,4136,8299,0,4369,8274,4374
,8297,4379,8298,4383,8289,4141,8296,4136,8299,0
,4374,8409,0,4374,8410,0,12403,12412,4369,8274
,4374,8297,4379,8298,4141,8296,4136,8299,0,4369
,8274,4374,8297,4379,8298,4141,8296,4136,8299,0
,12415,12416,12417,12418,12419,12420,12421,12432,4137,8417
,0,12435,12362,12327,12328,12338,12339,12371,12379,4374
,8419,0,4136,8420,0,12373,12377,12382,4374,8421
,0,12383,4353,8330,4358,8329,0,4136,8423,0
,4355,8424,12387,4142,8287,0,4369,8426,0,12385
,12378,4189,8427,0,4189,8428,0,4352,8252,4359
,8233,4363,8237,4365,8207,4367,8235,4368,8236,4369
,8211,4370,8229,4376,8225,4378,8232,0,12348,12347
,12341,12342,4155,8430,0,12350,12354,12351,12355,12352
,12356,4354,8350,12408,12410,12404,12405,4139,8356,4141
,8357,12414,4373,8367,4138,8365,4143,8366,12423,4373
,8367,4138,8365,4143,8366,12424,12426,12427,12428,12433
,4381,8432,0,4137,8433,4140,8434,0,12393,4137
,8435,4140,8436,0,12384,12375,4136,8437,0,4137
,8438,0,12391,12380,12381,12346,12343,12332,12392,12394
,12374,12376,12386,12388,4352,8441,4357,8442,4155,8440
,0,12333,4155,8443,0,12334,4352,8445,4155,8446
,0,4155,8447,0,12336,12335,-1};

int yypact[] {0,1,4,7,10,13,14,17,18,19
```

```
,38,61,64,65,86,87,88,93,112,113
,114,117,122,123,124,129,134,135,140,157
,162,165,166,167,172,175,176,177,180,195
,180,198,215,218,237,240,241,250,255,256
,277,278,283,284,287,310,319,322,323,324
,325,326,329,330,333,334,335,336,339,340
,343,344,345,346,349,350,367,350,370,387
,388,389,390,395,396,397,400,409,410,413
,414,419,422,423,426,431,432,435,436,447
,464,471,472,473,474,475,486,487,180,488
,489,492,495,496,497,498,499,500,501,504
,172,180,180,507,508,509,284,510,511,514
,525,532,537,560,581,582,583,586,593,594
,595,596,597,598,599,600,601,602,603,604
,180,605,606,611,616,621,622,623,624,637
,650,653,656,657,658,669,669,680,681,682
,683,684,685,686,475,475,475,687,688,691
,692,693,694,180,695,696,697,698,699,702
,705,706,707,708,711,712,717,720,723,726
,729,730,731,734,737,758,759,760,761,762
,765,766,767,768,769,770,771,774,775,776
,777,782,789,796,797,798,799,800,803,808
,809,814,815,816,819,822,823,824,825,826
,827,180,828,829,830,831,832,833,834,841
,842,180,845,846,851,854,855,-1};

int yyr1[] {0,1,1,2,3,5,6,4,4,4
,4,9,9,9,11,11,11,8,8,14
,14,12,12,12,12,12,12,12,12,12
,12,12,13,13,24,24,25,27,28,28
,28,28,22,22,29,34,34,34,34,7
,30,31,35,35,35,35,39,36,16,16
,16,16,38,38,38,40,40,40,40,10
,10,10,18,18,44,45,19,19,46,23
,47,47,15,15,48,48,50,51,51,49
,49,49,53,53,55,52,52,58,59,56
,56,37,37,57,54,61,61,17,17,20
,21,33,62,26,26,42,60,60,60,63
,63,64,64,65,65,66,66,68,68,68
,68,69,69,69,67,67,67,70,70,70
,70,71,71,71,71,72,41,41,43,43
,73,32,-1};

int yyr2[] {0,1,2,2,1,2,2,3,2,2
,3,1,2,3,1,1,2,1,2,2
,1,2,3,1,1,2,3,2,2,1
,2,3,2,3,3,3,2,2,2,3
,3,2,1,1,4,3,4,6,5,2
,2,2,2,3,3,4,1,2,4,3
,3,2,3,3,3,1,3,3,3,2
,3,2,1,1,3,2,2,3,2,2
```

```
,2,1,2,3,1,2,3,2,3,2
,2,2,3,3,2,1,2,1,3,1
,3,1,1,2,3,2,3,3,2,3
,3,1,2,1,1,2,3,3,1,1
,3,1,3,1,2,1,3,1,1,1
,1,1,1,1,1,3,3,1,3,3
,3,1,1,1,2,3,1,3,1,1
,2,1,-1};

int yygo[] {0,-1,1,-1,2,-1,3,3,7,-1
,35,0,4,27,76,74,76,76,76,-1
,16,-1,8,-1,9,53,131,-1,46,9
,48,53,132,131,203,-1,12,9,47,12
,58,16,64,42,113,48,123,53,133,131
,204,132,205,203,237,-1,13,12,59,16
,63,27,73,42,114,48,59,74,145,76
,63,132,59,203,59,-1,14,27,75,-1
,17,-1,18,46,122,131,122,-1,49,-1
,50,-1,51,29,80,-1,20,-1,21,-1
,22,-1,23,-1,24,-1,25,-1,26,-1
,27,-1,74,37,89,39,109,40,110,107
,177,120,183,121,184,182,226,240,247,250
,252,-1,147,-1,34,-1,42,-1,116,-1
,117,-1,118,45,119,-1,38,33,87,81
,149,119,182,-1,77,-1,239,-1,53,-1
,54,54,135,136,208,194,230,-1,199,54
,136,-1,84,-1,55,-1,85,8,28,9
,28,12,28,16,28,27,28,29,81,42
,28,48,28,53,28,74,28,76,28,85
,151,87,154,131,28,132,28,203,28,-1
,102,85,152,-1,93,85,153,-1,101,-1
,30,-1,31,-1,32,-1,36,125,185,-1
,126,-1,127,-1,189,-1,192,129,200,-1
,190,-1,129,-1,186,-1,130,-1,193,-1
,194,-1,196,-1,197,197,233,-1,91,-1
,187,-1,29,-1,90,157,215,-1,92,158
,216,-1,94,97,162,-1,96,163,219,-1
,98,-1,163,-1,169,164,220,165,221,-1
,99,104,176,173,222,174,223,175,224,-1
,100,-1,103,-1,39,-1};

int yypgo[] {0,1,3,5,7,11,21,23,25,29
,37,57,77,81,83,89,91,93,97,99
,101,103,105,107,109,111,113,133,135,137
,139,141,143,147,155,157,159,161,169,173
```

```
,175,177,211,215,219,221,223,225,227,231
,233,235,237,241,243,245,247,249,251,253
,255,259,261,263,265,269,273,277,281,283
,285,291,301,303,-1};

int nterms 53;
int nnonter 73;
int nstate 256;
char *yysterm[] {
"error",
"ADDRESS",
"AND",
"AT",
"BASED",
"BY",
"BYTE",
"CALL",
"CASE",
"DECLARE",
"DATA",
"DO",
"ELSE",
"END",
"EOF",
"GO",
"GOTO",
"ID",
"IF",
"INITIAL",
"INPUT",
"MOD",
"NUMBER",
"OR",
"OUTPUT",
"PROCEDURE",
"RETURN",
"STRING",
"THEN",
"TO",
"WHILE",
"NOT",
"UMINUS",
"==",
"<=",
">=",
"<>",
0 };

char *yysnter[] {
"$accept",
"program",
```

```
"module",
"module.name",
"simple.do.block",
"label.definition",
"simple.do.state.head",
"simple.do.statement",
"decpart",
"unit",
"ending",
"statement",
"basic.statement",
"if.statement",
"declaration",
"declaration.statement",
"procedure.definition",
"assignment",
"return.statement",
"call.statement",
"input.statement",
"output.statement",
"do.block",
"goto.statement",
"if.clause",
"true.part",
"expression",
"group",
"group.head",
"step.definition",
"while.clause",
"case.selector",
"simple.variable",
"replace",
"iteration.control",
"procedure.head",
"procedure.name",
"basic.type",
"parameter.list",
"procedure.name.head",
"parameter.head",
"variable",
"locator",
"constant",
"typed.return",
"untyped.return",
"call.head",
"goto",
"declaration.element",
"type.declaration",
"data.list",
"data.head",
"type",
```

```
"array.specifier",
"initial.list",
"array.head",
"variable.type",
"based.variable",
"variable.type1",
"variable.type2",
"restr.expression",
"initial.head",
"left.part",
"logical.expression",
"logical.factor",
"logical.secondary",
"logical.primary",
"arith.expression",
"relation",
"com",
"term",
"primary",
"sub.expression",
"subscript.head" };
*
```

VITA

The author was born in Hu-Nan Province of Mainland China on April 5, 1949. She moved with her family to Taiwan in 1949 before the communist regime took over the Mainland. She completed her elementary and high school education in Taichung, Taiwan, and earned her Bachelor of Science Degree in Sociology at the Tung Hai University in 1970. She worked as teaching assistant in the Computer Science department of the University of Tennessee, Knoxville during 1977-1979. She has been working as a research associate at the office of Institutional Research since 1980.