To the Graduate Council:

I am submitting herewith a thesis written by Joseph F. Hull entitled "CM: A C Cross-compiler for Microcomputers." 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.

Charles E. Hughes, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Chancellor
Graduate Studies and Research

B

CM:

A C CROSS-COMPILER

FOR MICROCOMPUTERS

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Joseph F. Hull

August 1980

3044861

Copyright by Joseph F. Hull 1980

All Rights Reserved

ACKNOWLEDGEMENTS

I would like to acknowledge the assistance given me by each of the members of my committee and especially Dr. Charles Hughes, my major professor. In addition, I wish to acknowledge the very kind assistance of Dr. Forbes Burkowski of the University of Manitoba who furnished me with the initial designs and a prototype for the CM compiler.

ABSTRACT

CM is a cross-compiler, written in C, that runs on any computer using the UNIX(R) Operating System and generates code for a Z80 micro-computer. It is designed to be easily modified to generate code for a wide variety of micro-computers with architectures similar to the Z80. Another design consideration is to provide a tool for teaching classes on compiler design and construction.

PREFACE

The idea of writing operating systems in a structured language is fascinating. With a portable compiler and an operating system written in that language, a good programmer becomes completely hardware independent. Such a language would straddle the traditional division between assembly languages and compiler languages because it would need features of both. It would, at a minimum, require the bit-twiddling abilities of assembly languages and the control structures and compactness of higher-level languages. The C programming language, as defined by Brian Kernighan and Dennis Ritchie in their excellent book by that name(8), is a structured, compiler language intended for writing operating system software.

Of course, writing a portable compiler is a large problem, but it can be cut down by restricting the range of machines to which it will be transported. In this case, the range is restricted to 8-bit machines with a small number of registers and a 16-bit address space. To further reduce the problem, restrictions were placed on the language accepted by CM. The restrictions chosen reflect the nature of the machines in the range selected; e.g., they have no floating point instructions, so CM does not accept floating point data items.

TABLE OF CONTENTS

CHAPTER		PAGE
1.	INTRODUCTION.	1
	1.0 General Introduction	
	1.1 Characteristics of Systems Programs	
	1.2 A Few Words on Structured Programming	
	1.3 High-level and Low-level Languages	
2.	COMPILER TECHNIQUES	8
	2.0 Current Trends	
	2.1 Parsing Strategies	
	2.2 Characteristics of LR(1) and LL(1) Languages	
	2.3 Top-down vs Bottom-up Parsing	
	2.4 CM vs Table-driven Parsers	
3.	THE CM COMPILER	15
	3.0 The C Programming Language	
	3.1 A User's Guide to CM	
	3.2 A Maintenance Manual for CM	
4.	THE CODE GENERATOR AND POTENTIAL TARGET MACHINES. .	46
	4.0 Introduction	
	4.1 Potential Target Machines	
	4.2 The Architecture of the CM Pseudo-machine	
	4.3 Assumptions About Assemblers	
	4.4 Functions Required By CM	
	4.5 The Architecture of the Z-80	
5.	THE NEXT STEPS.	62
LIST OF REFERENCES.		64
VITA.		67

LIST OF FIGURES

FIGURE		PAGE
3-0.	Automatic Type Conversions.	18
3-1.	Precedence and Associativity of Operators	24
3-2.	Syntax of Flow Control Statements	30
3-3.	A Define-table Entry.	40
3-4.	A Symbol-table Entry.	41
3-5.	A Structure-list Entry.	41
3-6.	A String-list Entry	42
3-7.	The Two Forms of Derivation-tree Entry.	43
3-8.	Contents of a Procedure's Stack Frame	44
4-1.	The Instruction Set of the CM Pseudo-machine. . . .	51
4-2.	Functions Required By CM For Code Generation. . . .	53

CHAPTER 1

INTRODUCTION

SECTION 1.0 General Introduction

In the world of computers, some needs are universal. Every computer needs programs to direct and control its peripheral equipment. In the early days of computing, all programmers wrote their own, but this was inefficient, so standard programs were created to ease the burden. Each computer installation kept a "library" of such programs for use by any of the programming staff who needed them.

During these early times, programmers wrote their programs in machine language, but this, too, was inefficient and the idea of program-generating programs (compilers and assemblers) was born. Also recognized was the need for a program that would load another program into the computer. As the number of programs increased, so did the difficulty in finding the program one wanted (Now, where did I put that card deck?), which led to the idea of a file system and file handler. Since every computer system needed these programs, programmers began to call them systems programs. The class of systems programs has been expanded to include job scheduling algorithms, resource allocation algorithms (both

physical device allocation and information storage allocation) and others. Systems programs may be distinguished from applications programs in that they are primarily concerned with access to the computer and its capabilities. Applications programs, on the other hand, are concerned with solving some problem or meeting some need that has no direct connection with computers, that is, a computer will be used if and only if it is an efficient or convenient tool for that particular application.

SECTION 1.1 Characteristics of Systems Programs

As systems programs grew larger and more complex, they became more difficult to implement and to maintain. Many authorities, notably Edgser Djikstra, have suggested that control structures offer a means of organizing a program such that modularity is increased and connections between modules become more obvious. This results in programs that are easier to understand and maintain. Another benefit is that large programs are easier to implement since the implementation may readily be separated into smaller programs or subprograms. These advantages apply to any large program, some say to any program larger than one page, whether it is a systems program or an

applications program.

The needs of systems programmers are slightly different from those of applications programmers. Those needs are met by some features of the programming language C that do not exist in most other programming languages. In systems programming, some operations occur with much greater frequency than they occur in applications programming. For example, increment or decrement by one is used extensively by I/O routines and lexical scanning routines. Such an operation (increment and/or decrement) would obviously be convenient for adjusting the value of a pointer, i.e., a variable whose value is the address of another variable. The increment operator would be even more convenient if it adjusted the value of the pointer by the unit size of the object pointed at rather than the arithmetic value one. And again, the construct I=I+ROWSIZE is used to access successive columnar elements of a multiply-dimensioned array. Since, in higher-level languages, ROWSIZE could be replaced by an arithmetic expression of arbitrary complexity, it is obvious that this construct leads to greater depth of parse tree than would an operator that combined the additive property of + with the assignment property. This is not to suggest that systems programming cannot be accomplished without such operators, but only that it is much more convenient with them. Similarly, ease of

parsing and convenience of programming can be obtained by providing the ability to assign a value to a variable at the same time that the variable is tested. One way to do this is to treat the assignment operator as an expression, that is, treat it like any other binary operator. Then, when an assignment expression is encountered, it is evaluated to yield a value, which value will be the same as the right-hand-side (RHS) of the expression.

SECTION 1.2 A Few Words on Structured Programming

It is not the purpose of this paper to discuss structured programming; yet it is unacceptable to use the phrase without some definition. Structured programming, then, as used here, means programming in a style that does not violate the following rules:

> (1) Programs should be subdivided into units, called modules, on the basis of function. Insofar as possible, a module should have a single function. Where it is necessary or convenient that a module should be more general (have more that one function), care should be taken that this fact is obvious.

(2) Modules will be clearly delineated and the connections between modules will be explicit.

(3) The flow of control shall enter a module at the top and, insofar as possible, exit at the bottom.

(4) Control of flow in a program shall be, as much as possible, through the use of these basic control structures, sequence, repetition and selection (i.e., avoid goto's). Blank lines and indentation should be used to provide visual indication of these control structures.

While more elaborate definitions of structured programming exist, this simple one should be adequate for the present purpose. It should be noted that this definition, while it encourages the use of procedures to implement the subdivision of a program, does not mandate any specific technique.

SECTION 1.3 High-level and Low-level Languages

Computer programming languages can be characterized as low-level or high-level. The low-level languages are,

primarily, the assembler lenguages or symbolic languages that correspond to each computer's instruction set. The high-level languages include FORTRAN, ALGOL, COBOL, PASCAL and others. The definitive difference between these two classes of languages is that, for any given program in a low-level language, there exists one, unique, correct translation into the machine language of the target computer while, for any given program in a high-level language, there exist many equivalent translations.

The low-level languages normally have a "one source statement to one machine instruction" translation (although macros are an exception to this) while, for high-level languages, one source statement may be translated into many machine instructions. The complexity of a single statement or expression permitted by the high-level languages is far greater than that permitted by the low-level languages and the difficulty of code generation is commensurately greater.

The low-level languages offer no type-matching features (a la PASCAL) or automatic conversion of variable type (as does PL/1). Type-matching is the recognition that two variables in an expression have been declared to contain different types of data, e.g., character vs. integer. Automatic conversion, then, is the coercion by the translator of one type to another so evaluation of the expression may continue, without intervention by the user.

The low-level languages provide no procedure invocation and argument passing mechanism, although one can be constructed from language elements present. Nor do they provide a space for local variables that are never used by other procedures.

Finally, unlike low-level languages, the high-level languages provide some or all of the control structures that simplify writing and maintaining programs.

CHAPTER 2

COMPILER TECHNIQUES

SECTION 2.0 Current Trends

Today, the thrust of compiler design seems to be directed toward the automatic and semi-automatic techniques, specifically LL(1) and LR(1) with emphasis on LALR(1) where the latter is chosen. Most researchers in the field seem fascinated by the fact that the tables for these table-driven parsers can be generated algorithmically with little or no human intervention. The choice of the recursive-descent technique for the CM compiler was based on two points: first, that recursive descent more closely follows the human thought process (at least, it more closely follows this author's thought process) and, second, that the space needed by the syntax analysis phase of table-driven LALR(1) parsers is larger than that required by recursive descent parsers(10). The first motivation is significant in that this project is primarily a learning experience for the author. The second assumes major importance for the class of potential target machines for which this compiler is intended.

SECTION 2.1 Parsing Strategies

Parsers are classified by the parsing strategy which they implement. Some order must be placed on the input to a parser and, since most of the work on parsing has been done by people who speak English, a left-to-right order has been chosen for all, or nearly all, parsers.

The action of parsing generates a derivation tree of the input scanned by the parser and the parser will be further classified by the characteristics of the derivation tree it generates. Current theory and practice suggest that two characterizations are most worthy of study and development: left-to-right scanning generating a rightmost derivation tree (LR), and left-to-right scanning generating a leftmost derivation tree(LL). These techniques are further characterized by the number of input tokens that must be scanned in order to identify the particular production of the underlying grammar that includes in the language being recognized any particular substring of the string being parsed. Thus, an LR grammar that requires k tokens to distinguish between any two productions of that grammar is called an LR(k) grammar. It has been shown that for any LR(k) grammar, G, there exists an LR(1) grammar, H, such that the language generated by G is precisely the language generated by H and likewise for LL(k) grammars.

Therefore, attention is focussed on LR(1) and LL(1) grammars.

SECTION 2.2 Characteristics of LR(1) and LL(1) Languages

In application to programming languages, certain characteristics of LR(1) and LL(1) languages assume a high degree of importance.

First, they are predictive. That is, for any given state of the parser and any input symbol, there is at most one production of the underlying grammar which can apply. A result of this property is the elimination of any need to backtrack during the parsing process, which means a reduction in the amount of memory and execution time needed by the parser. The order of this reduction may easily be several orders of magnitude for a large program.

Second, these languages display the valid prefix property. When parsing to determine if a string is included in one of these languages, it is possible to determine which token is responsible for the exclusion of the string from the language. That is, the parser can recognize where it should issue an error message, not just that the string is not in the language. Usually, the valid prefix property also means that the parser knows what the next token should

have been and can, therefore, issue an informative diagnostic message rather than, "Something is wrong here!"

SECTION 2.3 Top-down vs Bottom-up Parsing

Because of the pattern of generation of nodes of the derivation tree, productive parsing is often called top-down parsing and reductive parsing is called bottom-up parsing. Two common top-down parsing techniques are recursive descent and LL parsing. Common bottom-up techniques are shift-reduce parsing and LR parsing.

The nature of top-down parsing offers some advantages in the matter of error message generation. That is, because the derivation tree is being constructed from the root proceeding toward the leaves, there is, at the time an error is detected, a large amount of information about the structure of the program available. This information can be reported to the user and should be of great assistance in finding and correcting the error.

On the other hand, bottom-up parsing offers a similar advantage for error correction. If a bottom-up parser detects an error, it is very likely that the local derivation tree is near a leaf. Furthermore, it is likely that there are only a few productions possible at that

point, no matter what the input token. This means the parser can select a production and continue the parse with a high probability that the production selected is what the user wanted in the first place. Of course, an error message will be issued, but the fact that the parse can continue and detect errors later in the program should save the programmer considerable time.

SECTION 2.4 CM vs Table-driven Parsers

As mentioned in the introduction to this chapter, the current trend in compiler design is toward table-driven parsing techniques, such as LL(1) and LR(1). There is a fascination with these techniques because the parse tables can be generated automatically or semi-automatically from the grammar underlying a language. To each production of the grammar is linked a set of actions for the parser to take when that production is encountered during the parse. Since it is fairly easy to associate machine instructions with the productions of a grammar, the implementation of such a compiler is not overly difficult. Furthermore, since the procedural part of such a compiler is very simple, these compilers are naturally fast.

On the negative side, due to the table-driven nature

of these compilers and also to the normal method of production of those tables, there is often a deficiency in the error messages produced. This is especially regrettable because the tables offer a very high potential for precise error messages. The amount of work necessary to identify just which message should be produced for any particular position in the parse table, however, approaches that needed to write a compiler by one of the older methods.

Concerning the present work, the non-obvious nature of the parse process using tables is an unacceptable drawback. One of the goals of this project was to provide a compiler that could be used by future classes in compiler design and construction as an educational tool. To that end, the CM Compiler has been designed to be easy to understand. If the implementation of any function offered the choice between efficiency and confusion on the one hand or ease of understanding and inefficiency on the other, the latter was always chosen. In addition, flags were included to allow the user to display a trace of functions invoked during the parse, the code as it is generated, the source as it is read by the parser and the parse tree itself.

The second major point concerns the amount of memory required by table-driven parsers. Although CM is a cross-compiler, an obvious extension of the present work is enhancement of CM until it can compile itself, thereby

producing a compiler for a microcomputer that runs on that microcomputer. This is what Snyder calls the half-bootstrap process(10). In that effort, it will be crucial that the compiler use no more memory than absolutely necessary. Although the space occupied by the code of a LALR parser is much smaller than the space occupied by the code of a recursive descent parser, the total space occupied by the code and tables of a LALR parser is roughly equal that of a recursive descent parser, with the LALR parser being slightly larger. Add to that the additional space required by the larger portion of the derivation tree that is retained by the LALR parser during parsing and code generation and the balance is decidedly tipped in favor of the recursive descent parser.

CHAPTER 3

THE CM COMPILER

SECTION 3.0 The C Programming Language

C is a general-purpose programming language. It is a low-level language in that objects in C are the same as in most computer machine languages, i.e., addresses, characters and numbers. These objects can be manipulated with logical and arithmetic operators. They can be aggregated as arrays, structures and unions, but C does not provide any way to manipulate them as aggregates. The only storage allocation facilities provided by C are STATIC definition and a stack discipline for local variables or functions. There is no dynamic allocation or garbage collection. C does not include any input/output facilities. C offers single-thread control flow constructs but not multiprogramming, parallel operation, synchronization or coroutine primitives. Typically, these features are provided by explicitly-called functions when and where they are desired or needed.

The small size of the language, due to omission of high-level features mentioned above, makes C easy to learn and to implement. C compilers are, typically, simple and compact. They are easy to write and can be run on very

small machines. C is independent of any particular machine architecture, so portable programs can be written easily. At the same time, since the data items of C so closely match those manipulated by existing computers, C programs are usually efficient in terms of computer memory and execution time.

The rest of section 3.0 is a brief description of the C programming language. It is addressed to the knowledgable programmer who knows one or more high-level languages but does not know C. It is intended to give this reader enough information about the language to facilitate the later discussion of CM but it is not adequate as nor intended to be a text book. In the discussion that follows, quotation marks are used to emphasize words where there might be some ambiguity otherwise. Upper-case letters are used for descriptive text, or meta-language as it is sometimes called.

SECTION 3.0.1 Operands and Operators

SECTION 3.0.1.1 Variable Names

Variable names in C consist of letters and digits. Letters are the 26 upper case letters, the 26 lower case

letters and the underscore. Digits are the characters 0 through 9. Variable names may be of arbitrary length, although only the first 8 are significant. External names, e.g. functions or external variables, may have other restrictions.

SECTION 3.0.1.2 Data Types

The basic data types of C are "char", "int", "float" and "double". A variable of type "char" will hold 1 character of the local character set, typically 1 byte. A variable of type "int" will hold an integer with the size most appropriate to the machine in question. A "float" variable will hold a single-precision real number and a "double" will hold a double-precision real number with the precision being machine dependent. There are also 3 qualifications to type "int": "short", "long" and "unsigned". The intent is to provide different length integers for space optimization. Unsigned integers use modulo 2**n arithmetic, where n is the number of bits most natural to the particular computer.

Another data type is the pointer. This item contains the address of the memory location that contains a value of some type. The syntax of a pointer declaration is

```
TYPE *POINTER_NAME;
```

For example, a pointer to an integer would be declared

```
int *int_pointer;
```

The variable "int_pointer" would have type pointer-to-integer and "*int_pointer" would have type integer.

SECTION 3.0.1.3 Type Conversion

C limits automatic type conversion to cases which occur frequently in systems programming. Implicit arithmetic conversions for binary operators are described in Figure 3-0.

```
CONVERT CHAR AND SHORT TO INT
CONVERT FLOAT TO DOUBLE
IF      EITHER OPERAND IS DOUBLE
        CONVERT OTHER OPERAND TO DOUBLE
        RESULT IS DOUBLE
ELSE IF EITHER OPERAND IS LONG
        CONVERT OTHER OPERAND TO LONG
        RESULT IS LONG
ELSE IF EITHER OPERAND IS UNSIGNED
        CONVERT OTHER OPERAND TO UNSIGNED
        RESULT IS UNSIGNED
ELSE    RESULT IS INT
```

Figure 3-0. Automatic Type Conversions

Notice that all floating point arithmetic in C is done in double precision.

Conversion takes place on assignment. Remember that an assignment expression yields a value just as any other expression. The type and value of an assignment expression will be the same as its left operand with the right operand value being converted as necessary.

The conversion of a variable of type "char" to one of type int is machine dependent. On some computers, "char" is converted to an integer in the range [0,255]. On others however, sign extension takes place, yielding a negative integer when the high order bit is set. If a particular computer system uses the ASCII character set, the values will always lie in the range [0,127]. The conversion from "int" to "char", "long" to "int", or "long" to "char always truncates the high order bits. The conversion of "float" to "int" or "double" to "int" truncates any fractional part of the value while "double" to "float" rounds off. Since a procedure invocation is an expression, conversions of "char" and "short" to "int" and "float" to "double" occur automatically when arguments are passed.

A type conversion can be coerced in any expression using a cast. A cast is a construct of the form

```
(TYPE-NAME) EXPRESSION
```

For example

```
(int) c + `a`
sqrt((double) j)
```

The latter is equivalent to:

```
int j;
double t;
t=j;
sqrt(t);
```

except that the temporary storage used by the cast is available for use whenever the compiler recognizes the need for temporary storage while the variable t is used only when the programmer specifies it.

SECTION 3.0.1.4 Constants

An integer constant in C consists of a string of digits. If the value is too large to fit into an "int" or if the digits are followed by the character "L", the constant is taken to be "long". A "float" constant has 2 forms: a string of digits with a decimal point or a number in scientific notation. "Float" and "double" constants are stored as "double", so the same notation serves for both. A leading zero on an "int" implies that it is an octal constant while a leading 0x or 0X implies a hexadecimal constant. The digits 0 through 9 and the letters A through F, upper or lower case, are used as hexadecimal digits. A character constant is a single character written between

apostrophes, e.g. 'a'. Characters and character variables may participate in arithmetic operations. Escape sequences for character constants are provided with the backslash character. Some escape sequences are:

\\ - to generate the backslash itself

\' - the apostrophe,

\n - a newline

\t - a tab

\d - where d is 1 or 2 or 3 octal digits and generates a character-size bit pattern

A string constant is a sequence of 0 or more characters between quote marks, e.g. "a string constant". The escape sequence \" represents the quote mark and the escape sequences for character constants apply to strings. The quote marks delimit the string, they are not part of it. A string is treated as an array of single characters and the compiler will automatically place a null character, '\0', at the end of the string. The length of the sample string constant given above is 17 but it uses 18 bytes of memory space. The user should be careful to distinguish 'a' and "a" as they are not identical.

An expression, all of whose operands are constants, is called a constant expression. Constant expressions are evaluated at compile time and may be used wherever a constant may be used, e.g. in array dimensions.

SECTION 3.0.1.5 Declarations

All variables must be declared before use. A declaration specifies a type and a list of 1 or more variables of that type. A type may appear more than once. Variables may be initialized in the declaration. External or static variables are initialized once while automatic variables are initialized each time the function is invoked. External and static variables are initialized to 0 by default, but the preferred style is to give them explicit initial values. Automatic variables have undefined values if there is no explicit initial value given.

C offers two aggregate forms of variables, arrays and structures. An array is an aggregate in which each element is a variable of the same type as every other element. A structure is an aggregate in which elements may be of different types. Elements of an array are addressed by the array-name and a subscript, e.g., grades[4]. Members of a structure are addressed STRUCTURE_NAME.MEMBER_NAME or POINTER_NAME->MEMBER_NAME. C permits arrays of structures and structures that have arrays as members.

C also permits a block of memory to be used in more than one way. The declaration

```
union U_NAME {
        char  TYPE_1;
```

```
        int   TYPE_2;
        float TYPE_3;
    } U_VAR;
```

will reserve enough space for a "float" variable (the largest in this union) and permit the user to store any of the declared types in that space. Union members are addressed similarly to structure members, e.g., U_VAR.TYPE_2 = 5.

SECTION 3.0.1.7 Operators

The precedence and associativity of C operators is shown in Figure 3-1. Operators of the same precedence are shown on the same row and rows, top to bottom, are in decreasing order of precedence. Operators associate left to right, except those in rows marked RL, which associate right to left.

The binary arithmetic operators are "+", "-", "*", "/" and "%" (modulus or remainder). When "/" is applied to "int"s, any fractional part is truncated. The "%" operator cannot be used with "float" or "double". The order of evaluation for associative and commutative operators is not specified, even when expressions are parenthesized.

C provides logical and relational operators. To

```
     ()   []   ->   .
RL   !    ~    ++   --   -    &    *    sizeof   (type)
     *    /    %
     +    -
     <<   >>
     <    <=   >    >=
     ==   !=
     &
     ^
     |
     &&
     ||
RL   ?:
RL   =    +=   -=   *=   /=   %=   &=   |=   ^=   <<=   >>=
     ,
```

Figure 3-1. Precedence and Associativity of Operators

indicate a different (higher) precedence, "<", "<=", ">" and ">=" are called the relational operators while "==" and "!=" are called the equality operators. The logical connectives, "&&" and "||", have different precedence from each other and expressions using them are evaluated left to right, with evaluation ceasing as soon as the truth or falsity of the expression is recognized. The unary negation operator, "!", converts a true expression to a false one, and vice-versa.

The increment and decrement operators, "++" and "--", are unusual in that they may be applied as prefix or postfix operators. When the resulting value is not used immediately, either may be chosen, but, in an expression, the action of a prefix operator is quite different from a postfix. For example, after the sequence:

```
i = 4;
```

```
j = 7;
m = i++;
n = ++j;
```

the value of m is 4, i is 5, while j and n have value 8. Furthermore, if p is a pointer, then p++ increments p to point at the next element of whatever kind p has been declared to point at. After this sequence:

```
char *c;
int  *i;
c = 1000;
i = 1000;
c++;
i++;
```

c has the value 1001, while i has the value 1002, assuming characters occupy one byte and integers two bytes.

C's orientation toward systems programming shows in the presence of the bit manipulation operators. The unary one's complement operator and binary and, inclusive or, exclusive or, shift left and shift right operators (~, &, |, ^, <<, >>) may not be applied to "float" or "double" operands, but they are necessary for working with binary flags.

C provides a set of assignment operators unique among programming languages. In addition to the standard "=", C has "+=", "-=", "*=", "/=", "%=", "<<=", ">>=", "&=",

"|=", and "^=". The utility of these may be seen in the familiar i = i + j, which is written in C as i += j. The advantages are clear: less typing, no need to check that two expressions are identical, closer correspondence with human thought processes and, since the left hand side need be evaluated only once, more efficient code. The type of an assignment expression is the type of its left operand and an assignment expression has a value (just as any other expression).

In the interest of concise, unreadable code, the designers of C have included a ternary conditional operator, "?:", whose use is best explained by example. The following two code sequences are equivalent and determine if a character is a digit:

```
if ('0' <= c  &&  c <= '9')
        answer = 1;
else    answer = 0;
```

```
answer = ('0' <= c  &&  c <= '9') ? 1 : 0;
```

As most languages, C has many uses for parentheses. Primarily, they are used to change the order of evaluation in expressions, but they are also used to enclose the argument list of a procedure invocation and the parameter list of a procedure definition. One supposes these last two could be considered a special case of an expression and,

therefore, that all three are one use, but since C also uses parentheses to form a cast there would still be multiple uses. C does, however, make it easy to distinguish array references from procedure invocations by using square brackets, "[]", to enclose subscripts. Furthermore, C uses braces, "{}", to indicate a compound statement or block (a procedure is a named block) and in the definition of structures. Since a structure may be thought of as a "compound declaration," these are, at least, similar uses and there is no difficulty in distinguishing the constructs when either is encountered in a program.

Unless one is engaged in list-processing (singly-linked lists, doubly-linked lists, trees, queues, stacks, etc.) or some dynamic storage-allocation process, there is little use for address and pointer operators. But since systems programming abounds with such processing, C includes address of a variable, pointer to a variable, member of a structure and pointer to member of a structure operators (&, *, ., ->). The unary pointer operator, "*", is distinguished from the binary multiply operator, "*", contextually. The "." and "->" operators are very similar, p->b being shorthand for (*p).b where b is a member of structure s and p = &s.

The "sizeof" operator is unique to C, but why? It provides access to the size of a variable or aggregate item

and is very convenient is both systems and applications programming.

C also includes the "," operator which guarantees that expressions separated by it will be evaluated left to right with the type and value being that of the right operand. Note that the commas which separate variable names in a declaration list and those which separate procedure arguments are merely syntactic tokens, not comma operators.

SECTION 3.0.2 Block Structure

A block in C consists of zero or more declarations of variables followed by zero or more statements that use the variables known in that block. A block is enclosed in braces, "{}". A block may be contained within another block. In this case, any variables known in the outer block will be known in the inner block, unless the inner block contains a declaration for a variable of the same name, in which case, only the "local" variable will be known. A block is syntactically equivalent to a single statement.

A function in C consists of the function name followed by the (possibly empty) argument list enclosed in parentheses followed by declarations for the variables of the argument list followed by a block. In C, the word

"function" encompasses both procedures that return values and those that do not. The function named "main" is invoked by the operating system of the computer and control is passed back to the operating system on exit from main.

SECTION 3.0.3 Flow of Control

In C, as in other programming languages, the basic flow of control through a program is from one statement to the next, with the order defined as left to right on a line and from the top line of a program to the bottom. C possesses a variety of constructs to alter this basic flow of control. While C does include a goto statement, most of these constructs implement the structured programming concepts of repetition and selection. The syntax of these statements is shown in Figure 3-2.

SECTION 3.0.3.1 Repetition

The "while" statement offers a program loop with the test for terminal condition being made before the loop is executed. The "do" statement is a program loop with testing after the body of the loop has been executed. Neither statement provides for modification of the loop control

```
while (EXPRESSION)
        STATEMENT

do
        STATEMENT
while (EXPRESSION)

for (EXPRESSION_1; EXPRESSION_2; EXPRESSION_3)
        STATEMENT

if (EXPRESSION)
        STATEMENT_1
else
        STATEMENT_2

switch (EXPRESSION) {
        case VALUE_1:    STATEMENTS
        case VALUE_2:    STATEMENTS
        .
        .
        .
        case VALUE_n:    STATEMENTS
        default:         STATEMENTS
}
```

Figure 3-2. Syntax of Flow Control Statements

expression.

Iteration is a special case of repetition and is provided by the "for" statement. The "for" statement is equivalent to:

```
EXPRESSION_1;
while (EXPRESSION_2) {
        STATEMENTS
        EXPRESSION_3;
}
```

and choice between this and the "for" statement is a matter of style. Normally, EXPRESSION_2 is a relational

expression, but, since C conventionally treats a zero value as FALSE and non-zero as TRUE, it can be almost any expression. None of the expressions is required and if EXPRESSION_2 is absent, it is treated as permanently TRUE. C treats iteration differently than other languages in that the limits of iteration can be changed from within the loop, indeed, the value of any or all variables in the loop control expression can be altered. Also, the controlling variable(s) retain their values after termination of the loop.

It should be noted that for loops are not restricted to arithmetic progressions. Because the components are arbitrary expressions, a "for" statement is quite capable of, for example, list processing.

SECTION 3.0.3.2 Selection

The primary selection statement in C, as in other programming languages, is the "if" statement. The "else" clause is optional. C also provides the "switch" statement,

which is equivalent to:

```
if (EXPRESSION == VALUE_1) goto LABEL_1
else if (EXPRESSION == VALUE_2) goto LABEL_2
.
.
.
else if (EXPRESSION == VALUE_n) goto LABEL_n
else goto LABEL_n1;
LABEL_1:        STATEMENTS
LABEL_2:        STATEMENTS
.
.
.
LABEL_n:        STATEMENTS
LABEL_n1:       STATEMENTS
```

SECTION 3.0.3.3 Early Termination of Loops

Recognizing that the ability to exit from the middle of a program loop makes structured programming much more convenient, the designers of C provided two statements for this purpose. The "break" statement transfers control immediately to the statement following a "switch", "while", "do" or "for" statement. The "continue" statement, which

does not apply to the "switch" statement, causes execution of the next repetition of a program loop. That is, a "continue" statement causes an immediate transfer of control to the incrementing part of a "for" statement (i.e. EXPRESSION-3) or to the testing part of a "do" or "while" statement.

SECTION 3.1 A User's Guide to CM

CM runs on a DEC PDP-11 under the UNIX (R) Operating System. The executable file occupies some 32K bytes of disk storage; the source files, 70K bytes. Using CM to compile a program is simplicity itself: you type a command line to UNIX, giving the name of the file that contains the program you wish to compile, e.g.,

```
cm test.c
```

The source file name need not have the extension ".c." The CM compiler always generates a file named cm.out to contain the assembly language program it produces.

There are several compile-time flags that may be set but none of them affects the code produced by the compiler. All of the flags presently implemented display various tables or other internal data values during compilation. Some simply offer a convenient means of doing something that

could be done another way. All of these flags work on a hierarchical scheme such that if and only if the current value of the flag exceeds a preset level will that particular display be produced.

These flags are specified in the standard UNIX fashion of preceding them with a dash on the command line, e.g.

cm -cd8p test.c

There is an internal flag value (named debug) that defaults to one. In the example above, the c flag is set to one and the p flag is set to eight.

SECTION 3.1.0 CM Compiler Options

The first flag, "c", causes CM to list on the terminal the code written to cm.out. The level above which this display will appear is zero.

The second flag, "d", changes the value of the internal flag value. In the example above, the internal flag is set to eight. Due to the simple method used to convert the ASCII '8' to an integer, only single digits will be treated correctly. Presently, the maximum level used by CM is seven which means that eight is the largest number needed.

The 'l' flag causes CM to produce a source listing on the standard output device. The level for 'l' is zero. The 'm' flag causes CM to print the entire cm.out file at the termination of the compile step. The level for 'm' is zero.

The 'p' flag displays a trace of the procedures invoked during the parsing of a program. This display includes the line number of the source program statement currently being parsed, the name of the function invoked and some indication of the current token being parsed. This is the only display that really uses the level to affect the display. The larger the level number, the more information displayed. At trace level one, only the major parsing functions are traced. At trace level eight, every procedure is traced, including service routines.

The 'x' flag requests a display of the derivation tree. Level one displays derivation-tree entries as they are created. Level two displays them at code generation time. Since these two processes overlap in time, different information is presented, although the format of the display is identical. Any level above two will display entries at both creation and generation.

The last flag, 'u', causes a trace of the user functions of the code generator. These functions are so designated because they have no contact with the parser

except through other functions that are part of the code generator. They are separated from other functions which the parser invokes by name to permit the user to make any changes desired without fear of negative side effects on the remainder of the compiler. The levels for the 'u' flag should be similar to those for the 'p' flag, but such are not presently implemented.

SECTION 3.1.1 Language Restrictions of CM

CM does not implement all of the features of the C programming language. None of the features omitted are felt to be essential to a first implementation of C on micro-computers.

CM was developed on a system running UNIX(R) level six. The most notable change in the C language between levels six and seven was in the assignment operators. In level seven, that described in (8), the assignment with increment operator is "+=", while in level six, it was "=+". Likewise for the other assignment operators. CM accepts and is written in the level six version of C.

CM does not accept the "#include" compiler control statement of C. Since UNIX provides the "cat" command for concatenating files, this cannot be felt to cause any true

difficulty. In the extension of CM to run on the micro-computers for which it compiles code, this feature would need to be completely re-written, being so completely operating system dependent as it is. Therefore, its exclusion from the current implementation. Since CM does not offer the "#include" option, there was no need to provide the "external" type-modifier (see reference (8) for a description), so it was eliminated.

CM does not accept the "static" storage class designator. Global variables in CM are initialized once, at the start of a program. Local variables are initialized explicitly on entry to a procedure. It is felt that, since any function may access global variables, they can be used for those cases where a variable is required to retain a value over successive function invocations. Since local variables are allocated on the stack, some means of forcing static variables into the global area would be required anyway, thus the decision to make that fact obvious to the user by eliminating the "static" designation.

CM does not accept the "register" designation. Since potential target machines have relatively few registers, it is felt that they should not be pre-allocated in ways which may be severely detrimental to the overall efficiency of the program. In other words, applications programmers, keep your hands off. Even UNIX C does not

guarantee that "register" variables will be stored in registers, the "register" designation is a request, not a command.

There are no "float" or "double" variables in CM. None of the potential target machines known to the author have any floating point instructions, so floating point arithmetic will have to be simulated. Therefore, floating point arithmetic is a candidate for omission in this first implementation. It is an obvious candidate for the first round of improvements and its inclusion should be straightforward.

SECTION 3.2 A Maintenance Manual for CM

SECTION 3.2.0 Data Structures Used by CM

The major data structures used by CM are linked lists and trees. Since C offers the structure variable and pointer variable, these structures may be implemented directly, without resort to such artifices as parallel arrays.

The primary lists of CM are those for #define, symbols (i.e. variable names), structures and character strings. These are all singly-linked lists. CM uses trees

for the derivation tree and for temporary storage during the code generation process. Actually, these are the same tree with the contents of a node changed slightly.

SECTION 3.2.0.0 The Define Table

The define table contains one entry for each "#define" in a program. (Recall that the "#define" associates a label with a value and that, as a program is compiled, each time the label appears, it is replaced by the value, achieving the same result as though the value had appeared at each place in the program.)

The first element of a define-table entry, Figure 3-3, is the pointer to the succeeding entry. The second is the character string that appears in the #define statement while the last is the constant that will replace the defined identifier in the compiled code. The third element of a define-table entry records the type of the replacement value as either a number or a character string. It should be noted that, in the case of a defined identifier being replaced by a character string, defval is actually a pointer to the string.

```
struct defte { /* define-table entry */
        struct defte    *pdefte;/* link */
        char            *dname; /* identifier */
        int             defkw;  /* NUM or CHR */
        int             defval; /* value of entry */
}
```

Figure 3-3. A Define-table Entry

SECTION 3.2.0.1 The Symbol Table

The symbol table contains an entry for each symbol, or variable name, used in the program, except that structures are handled by a separate table. Like the define-table entry, the symbol-table entry, Figure 3-4, contains a pointer to the next symbol-table entry and a pointer to the character string that is the identifier. The third element of a symbol-table entry concerns the method by which CM allocates space for variables. All variables, local and global, are located by the distance they lie from a reference point. For globals, the reference is a fixed point in memory, the end of the code section of the program. For locals, the stack-frame pointer is the reference point. The value of the offset from the reference point is stored in vloc. The last element of a symbol-table entry tells whether the identifier is an integer, character, etc.

```
struct ste { /* symbol-table entry */
        struct ste      *pste;  /* link */
        char            *vname; /* identifier */
        int             vloc;   /* label/offset */
        int             *type;  /* see TY codes */
}
```

Figure 3-4. A Symbol-table Entry

SECTION 3.2.0.2 The Structure List

The structure list consists of one entry for each structure declared in the program. Each element, Figure 3-5, in the structure list contains a pointer to the next element of the list, a pointer to the name of the structure, the size of the structure and a pointer to a list of the elements of the structure, called the field list. Elements of the field list are the same as symbol-table elements.

```
struct strc { /* structure entry */
        struct strc     *pstrc; /* link */
        char            *sname; /* identifier */
        int             ssize;  /* structure size */
        struct ste      *field; /* field list */
}
```

Figure 3-5. A Structure-list Entry

SECTION 3.2.0.3 The String List

The string list contains entries for two types of data items, labels (targets of goto statements) and strings. The elements of a string-list entry, Figure 3-6, are a

pointer to the next entry, which type of entry, a pointer to an internal label for the string or goto label and a pointer to the characters of the string or goto label.

```
struct strb { /* string-list entry */
        struct strb     *pstrb; /* link */
        int             stype;  /* STR or ID */
        int             strlbl; /* internal label */
        char            *pstr;  /* string or goto label */
}
```

Figure 3-6. A String-list Entry

SECTION 3.2.0.4 The Derivation Tree

The elements of a derivation-tree entry, Figure 3-7, are a pointer to the type-vector of the entry (i.e. the type of the result when the expression is evaluated), the operator for this entry, a pointer to the left operand and a pointer to the right operand. For the address operator (&1) and the pointer operator (*1), both unary operators, the left operand is a special null value (NIL). For other unary operators, the right operand is NIL.

There is an alternate form of derivation-tree entry that is used by the code generator. This form relies on an integer being no larger than a pointer. In it, the first two elements are integers and the values placed there by the parser are not altered. The third element is an integer and is the value of the entry. The fourth is also an integer

and is the type of the entry.

```
struct xpr { /* expression-tree entry */
        int         *xtype; /* (pointer to) type vector */
        int         oper;   /* operator code */
        struct xpr *lopnd; /* (pointer to) left operand */
        struct xpr *ropnd; /* (pointer to) right operand */
}
struct uxpr { /* user-only expression-tree entry */
        int      r_[2];  /* preserved by user */
        int      value;
        int      uxtype; /* type code of value */
}
```

Figure 3-7. The Two Forms of Derivation-tree Entry

SECTION 3.2.1 Subroutine Linkage Conventions

Since programs compiled by CM are intended to be executed on machines with a simple architecture, a simple linkage convention is used. Since the target machines have only one register to be considered, the code generator already determines if the register contains a value that may be needed later and, if so, it stores the value in a temporary memory location. This involves no additional code because the logic used is the same as is used in evaluating arithmetic expressions. The intermediate level routines (load, assign, binoper, etc.) set the necessary flags and pointers to ensure no operand is lost.

Since C permits fully recursive procedure invocations, a stack frame is used for arguments and local

variables. This is very similar to the Procedure Activation Record of PASCAL(7). The contents of a stack frame are shown in Figure 3-8.

```
Arg n
Arg 0
# of bytes of arguments
Return address
Invoker's stack frame pointer
Local 0
Local n
```

Figure 3-8. Contents of a Procedure's Stack Frame

The stack frame is implemented using the Z-80's IX index register as the stack frame pointer. Since the Z-80 can address 256 bytes using an index register (-128 to +127), this implementation is limited to 122 bytes of arguments or 61 arguments (since each argument, even a character, uses 2 bytes). This implementation is also limited to 128 bytes of local variables but here variables use only the space they actually require.

Space for the stack frame is acquired from the Z-80's machine stack and stack discipline is maintained. Arguments are PUSHed onto the stack by the invoking procedure along with the number of bytes of arguments and the return address. On invocation, a procedure first PUSHes the invoker's stack frame pointer and then adjusts the stack pointer to provide enough space for all of its local

variables. On exit, a procedure uses these values to restore the stack to its previous state and reset the invoking procedure's stack frame pointer.

An alternate method of implementing an argument and local variable space for fully recursive procedures is also available. At compile time, fixed memory locations are allocated for both arguments and local variables for each procedure. This space holds the "current generation" of arguments and locals. When the procedure is invoked, the contents of this space should be PUSHed onto the machine stack and the "new generation" of variables moved into the space. This method does not require an index register and the machine stack remains a true stack. Access to an argument or local variable will be faster than when an index register is used. But a procedure invocation will require much more time because each element of the local space must be loaded into a machine register and then PUSHed onto the stack and later restored in like fashion.

CHAPTER 4

THE CODE GENERATOR AND POTENTIAL TARGET MACHINES

SECTION 4.0 Introduction

The lexical analysis phase of most contemporary compilers generates some form of intermediate code or language that is then acted on by the code generating segments of the compiler to produce the final output. Normally, this final output is the machine language of the target machine. One exception to this is the UCSD PASCAL compiler which produces P-CODE, the machine language of the PASCAL pseudo-machine. C compilers traditionally produce assembly language for the target machine rather than machine language and CM follows this model.

The study of intermediate languages and their generation is a complex issue beyond the scope of this paper, but most of them are produced through some form of syntax-directed translation. The CM compiler avoids the issue by generating code directly from the derivation tree at convenient times. This approach is facilitated by the CM pseudo-machine, a conceptual machine that implements, at the machine level, many of the operators of C. In other words, the architecture of the CM pseudo-machine was designed with

two primary goals: (1) to reflect the capabilities of currently available micro-computers and (2) to easily implement the operators and operands of the C programming language.

SECTION 4.1 Potential Target Machines

The development of computer hardware in the last 15 years is incredible. In 1967, when the author was programmer for a small college in Florida, the IBM 1130 was a hot, new machine. With 32K-bytes of memory, an operator's console, a line-printer, a card reader and one 5M-byte disk; it retailed for over $58,000. Today, for a comparable system, one might choose a Z-80 with 64K-bytes of memory, a video terminal, a dot-matrix printer and four 8-inch floppy disk drives (2M-bytes), which can be purchased for under $6,000.

The development of software for these new machines has lagged behind that of the machines themselves. Even five years ago, there was a notable lack of operating systems and compilers for these very small computers. Today, although there are a variety of operating systems on the market, they are relatively simple and not very powerful. The language picture is much better, with PL/M,

FORTRAN, PASCAL, C, COBOL, BASIC and others presently available.

Rather than attempt a comprehensive list of suitable machines that are currently available, a discussion of what constitutes a suitable machine will be presented.

The primary functional requirement CM places on a potential target machine is some manner of implementing indirect addressing. This could be accomplished by a self-modifying program that stores the address of the target in a subsequent instruction, but that is not al all a reasonable approach.

The next most important characteristic is a register at least as wide as the address space of the machine. Since most currently available machines in the class intended as targets of CM have 16-bit address spaces, this size is assumed throughout this paper. There are, in the sections that follow, places where that assumption may be ignored or modified if a machine with different architecture becomes available. Of course, registers of any size may be simulated by appropriate software if one is willing to pay the penalty this approach imposes in added processing time.

CM also requires conditional jump instructions. The target machine must have or be able to simulate six conditional jumps (equal, not equal, greater than, not greater than, less than, not less than) for both signed and

unsigned integers.

Other requisites can be extracted from the discussion of the CM Pseudo-machine, but they are either commonly available or easy to simulate and, so, will not be mentioned here.

SECTION 4.2 The Architecture of the CM Pseudo-machine

The CM pseudo-machine has one 16-bit register, QA. Some instructions operate in such a fashion that it is convenient to think of this register as two eight-bit registers with Q the high-order byte and A the low-order. The machine also has an internal condition code register (CONDITION) that may be queried as either signed (SIGNED_CC) or unsigned (UNSIGNED_CC) depending on which instruction is used. The instruction set of this machine was designed to have a close correspondence with CM operators. All instructions except the transfer instructions place appropriate values in the condition code register.

The C language is used as the basis for describing the instructions of the CM Pseudo-machine. In Figure 4-1, one-byte storage operands are designated by S, two-byte operands by SS. Two-byte operands are stored with the low-order byte at location X and the high-order byte at location

X+1. Storage operands may be prefixed by nothing, '#', or '*'. Nothing means the operand is the address of a value. '#' means the storage operand is immediate, i.e., part of the instruction. '*' means the operand is the address of the address of a value, i.e., an indirect address.

SECTION 4.3 Assumptions About Assemblers

Some equivalent of the following assembly-time mnemonics are assumed to exist:

adr place operand in next 2 bytes, low byte first

byt place operand in next byte

equ define label as equal to operand
('*' is the location counter)

res reserve, but do not fill, the next 'operand' bytes

In addition, it is assumed that there exists some character that will cause the assembler to treat the remainder of a line as comment text. A character variable named "comchar" must be declared and initialized to the comment character. Further, a character variable named "lblchar" must be declared and initialized to any value of the user's choice. This character will be used by CM to prefix all internally-generated labels and names.

```
OP      OPND    FUNCTION

add     S       / QA =+ S;
add2    SS      / QA =+ SS;
and     S       / A =& S, Q = 0;
cmp     S       / A - S;
cmp2    SS      / QA - SS;
dec     S       / S =- 1;
dec2    SS      / SS =- 1;
div     S       / QA =/ S;
get             / A = getchar(), Q = 0;
hlt             / exit();
inc     S       / S =+ 1;
inc2    SS      / SS =+ 1;
jeq     S       / if (  CONDITION == 0) goto S;
jge     S       / if (  SIGNED_CC >= 0) goto S;
jgt     S       / if (  SIGNED_CC >  0) goto S;
jhi     S       / if (UNSIGNED_CC >  0) goto S;
jhs     S       / if (UNSIGNED_CC >= 0) goto S;
jle     S       / if (  SIGNED_CC <= 0) goto S;
jlo     S       / if (UNSIGNED_CC <  0) goto S;
jls     S       / if (UNSIGNED_CC <= 0) goto S;
jlt     S       / if (  SIGNED_CC <  0) goto S;
jne     S       / if (  CONDITION != 0) goto S;
jmp     S       / goto S;
jsr     S       / s();
ld      S       / A = S, Q = A<0 ? -1:0;
ld2     SS      / QA = SS;
ldi             / A = *QA, Q = A<0 ? -1:0;
ldi2            / QA = *QA;
mul     S       / QA =* S;
neg             / QA = -QA;
or      S       / A =| S;
put             / putchar(a);
rem     S       / QA =% S;
rts             / return;
shl     S       / QA =<< S;
shr     S       / QA =>> S;
sto     S       / S = A;
sto2    SS      / SS = QA;
sub     S       / QA =- S;
sub2    SS      / QA =- SS;
xor     S       / A =^ S, Q = ~Q;
```

Figure 4-1. The Instruction Set of the CM Pseudo-machine

This character will be used by CM to prefix all internally-

generated labels and names.

SECTION 4.4 Functions Required by CM

Figure 4-2 lists the functions required by CM for code generation. A few of these are self-explanatory, the rest require some discussion.

In Figure 4-2 and the discussion that follows, "operand" refers to the current node of the derivation tree. "Right" and "left" refer to the right and left children of the current node. "Operand_type," "right_type" and "left_type" are boolean values that indicate if the object pointed to by that node can be stored in one byte or not. "New_node" is a derivation-tree node for which space has been allocated on the heap by the invoking procedure.

Throughout the code generation phase of any compiler, some indication of the contents of registers must be kept, along with an indication of whether or not that value is still needed. In the current implementation, space made available in the derivation tree by reduction of sub-trees is used to store the flags and pointers necessary for this bookkeeping. Other methods are possible and choice among them is largely a matter of style. The interested reader can find discussions of these in Compiler

```
addrof(operand)
assign(left,right,left_type,right_type)
binadd(left,right,left_type,right_type)
binand(left,right)
bindiv(left,right)
binmul(left,right)
binor(left,right)
binrem(left,right)
binshl(left,right)
binshr(left,right)
binsub(left,right,left_type,right_type)
binxor(left,right)
call(left,operand,number)
cmpa(left,right,left_type,right_type,op,flag,break_label)
cmpl(left,right,left_type,right_type,op,flag,break_label)
condff(operand,d,operand_type)
condtt(operand,operand_type)
copy(operand,new_node,operand_type)
deref(operand)
dupli(operand,new_node)
equgbl(label,offset)
equlcl(label,offset)
epilog(label)
incoper(left,right,left_type)
jmp(label)
jmpcase(switch_variable,value_table,address_table)
label(label)
outadr(label)
outchr(character)
outint(integer)
prolog()
purgtmp()
retval(operand)
unycom(operand)
unyneg(operand)
```

Figure 4-2. Functions Required By CM For Code Generation

Construction: An Advanced Course(13) and elsewhere. For the reader who wishes to attempt modifying CM for his own use, personal communication with the author will yield a more complete discussion than appears here. At any rate, the current implementation contains a function named "keep" that

marks the current contents of the register as needed.

The function named "addrof" implements the C address-of operator (&). The result of the operation is "kept", i.e., the "keep" function is applied to the result. Since the parser differentiates between the value and the lvalue of a variable, the function of "addrof" is to check that the current node is a pointer and, then, to change the type of that node.

Assignment of a value to a variable is handled by the function named "assign". The right operand is loaded, the appropriate store instruction is generated and the housekeepping flags are set. The argument left is "kept".

The binary arithmetic and logical "binadd," "binand," "bindiv," "binmul," "binor," "binrem," "binshl," "binshr," "binsub" and "binxor," all "keep" the left argument. Since integers are eight bits, only "binadd" and "binsub," which must also handle pointers, are capable of processing 16-bit operands.

The function named "call" implements procedure invocations. Prior to invoking "call", the parser issues appropriate invocations of "load" and "push" for the arguments of the procedure. "left" is a derivation-tree node that points to the name of the procedure to be invoked, "operand" is the node with the call operator and "number" is the number of bytes of arguments that have been pushed onto

the stack by the invocation. This last argument is necessary because C has no requirement that the number of arguments in a procedure invocation be the same as the number of parameters in a procedure declaration. The value returned by the procedure is "kept" in operand.

The logical and arithmetic comparison operators are handled by "cmpl" and "cmpa," respectively. The argument "op" tells which comparison to apply, "flag" tells whether or not the comparison should be inverted for ease of code generation. For example,

```
if (j == 5)
        b = 7*c;
```

should generate a compare followed by a jump on the not-equal condition to the end of the code generated by this statement. For this invocation of "cmpa," "left" would point to the node for j, "right" would point to a node for 5, "op" would be '==' and "flag" would be 0. The argument "break_label" indicates the point in the generated code that would be the target of a break statement.

The two functions, "condff" and "condtt," are used by conditional expressions, the ?: operator. "condtt" is invoked to load the true value and "condff" the false. The extra argument of condff points to the result node generated by condtt. In the current implementation, this is a null node, but future enhancements in the area of code

improvement will see a usable value being generated. Also, condff "keeps" its result.

The two functions, "copy" and "dupli," are similar in that they produce a new derivation-tree node with the same values as an already existing node. They are different in that copy then loads the new node and returns the "keep" of that node while dupli checks to see if the current register contents need to be saved in temporary storage. Both return a pointer to the new node.

"deref" operates directly on the derivation tree. In this implementation, the only action taken by deref is to load the variable pointed to by a pointer or array reference. Other actions are essentially null. The validity of a pointer is tested in that nodes of incorrect type will take the default branch of the switch.

"equgbl" and "equlcl" should generate labels for global and local variables, respectively. In the current implementation, global variables are generated as offsets from a common base address. This base address is just higher than the code segment. Local variables are all offset from the stack frame pointer. The parser keeps track of both offsets.

"epilog" is invoked only when the end of the source file is reached. The argument is the number of bytes required by all global variables used and epilog should

reserve enough space for them. In the current implementation, epilog is also responsible for generating code for handling certain standard functions of C that are best treated as subroutines rather than in-line code, e.g. the switch statement, return.

The increment and decrement operations occur so frequently in C programs that it is well to implement them very carefully. Both the ++ and += forms are handled by the same code, indeed, the parser reduces j++ to j += 1. The function "incoper" generates code for increment and decrement operations. The current implementation generates the appropriate number of "inc" or "dec" instructions. An obvious improvement is to place a limit on the number of "inc" instructions that may be generated, using an "add" of the appropriate constant for increments over that limit.

"jmp" is a difficult function to describe. The first thing that must be said is jmp does not generate jump instructions, it merely sets values for labels that will later be the target for jump instructions. There are three labels to be considered here. The first, "curlbl," holds an outstanding label request generated by the function label. The second, "jmplbl1," is the target of a control-structure jump such as an if statement might cause. The last, "jmplbl2," is the target of a second control-structure jump such as that required at the end of the then clause of an if

statement. In the current implementation, "jmpcond" and "xoutput" work with jmp to improve the code generated by CM by eliminating jump instructions whose target is a jump instruction. jmp is invoked by the parser to handle break and continue statements and other times when a branch to the end of code generated for a statement is required.

"jmpcase" generates code to handle a switch statement. That is, if a program contains a switch statement, CM will automatically include code that compares the switch value with values in a table. If there is a match, CM will cause a branch to the appropriate address stored in a matching table. If there is no match, CM will cause a branch to the address that is last in the table. jmpcase causes the appropriate values to be loaded for this common code to use.

The function "label" is part of the jump-jump improving code. It sets or changes outstanding jump labels and generates an "equ" pseudo-op for its argument.

The functions "outadr," "outchr" and "outint" each generate memory initializing a pseudo-op for the appropriate object. For example, "outchr('A')" would generate "byt 65," 65 being the value of the letter A. Remember that integers are eight bits.

"prolog" initializes the jump labels, housekeepping flags, etc. and generates a subroutine call to the function

named main. CM, like so many other compilers, treats the main program as a subroutine of the operating system. If there are arguments, it pushes them, just as does a procedure invocation. Finally, it generates a return to the operating system.

"purgtmp" purges the outstanding temporary storage requests. It sets to zero the number of bytes of temporary storage needed and resets the housekeepping flag that says the register contents are needed.

"push" implements the stack operation of that name. If the target machine has a "push" instruction, as does the Z80, then "push" need do no more than generate it. Otherwise, it will be necessary to simulate a stack within the instruction set of the target machine. CM does not require that stack frames be placed on the expression evaluation stack although the current implementation does just that.

"retval" implements the C return statement. If its argument is not null, it generates a load for that value. Then it generates a return instruction.

The unary operators for complement (one's complement) and negate (two's complement) are treated as binary operators with a null right operand. The function names are "unycom" and "unyneg."

SECTION 4.5 The Architecture of the Z-80

The Zilog Z-80 was chosen as the target machine for the current implementation of CM. The primary reason for the choice was availability. But the Z-80 might well have been chosen even if the author did not own one. It is a powerful micro-computer that is quite representative of its class of computers. It is a very popular machine, relatively inexpensive and readily available.

The Z-80 has an 8-bit accumulator and an 8-bit condition register. It has three 16-bit registers which can be accessed as six 8-bit registers. These can be incremented, decremented and used for indirect addressing. One of them can be used for 16-bit arithmetic (add and subtract only). It has a second set of registers identical to the ones just described (but only one set may be used at a time). It has two index registers that can be used to address 256-byte blocks of memory and can be incremented or decremented. It has a stack pointer.

The Z-80 instruction set is richer than most similar machines. It has push and pop instructions to manipulate the stack. It has an unusually varied set of load and store instructions. In addition to conditional jump instructions, it has conditional call instructions that provide subroutine linkage via the stack. It has load, store, compare, input

and output instructions that increment or decrement certain registers as a side effect of their more common functions. All of these are in addition to the add, subtract, and, or, exclusive or and bit-twiddling commands common to this class of machines.

CHAPTER 5

THE NEXT STEPS

There are several obvious extensions to CM. First, it will be desirable to enhance CM until it can compile itself, then re-write the operating-system-dependent portions and move CM to the microcomputer for which it compiles code. This is what Snyder(10) calls the half-bootstrap process. This process will take several steps, the most important being the inclusion of 16 and 32-bit integers and arithmetic routines to manipulate them.

Another extension is the enhancement of existing displays and addition of new displays of compiler internal data structures. There is also some maintenance work to be done on the internal documentation of CM, both commentary text and choice of data names. Some rearrangement of the order in which functions are presented will probably be desirable.

The code generator can be improved considerably. There are some code sequences which can be improved and the addition of code improvement routines, data flow analysis, common-tail analysis, etc., is highly desirable.

Finally, there are other microcomputers for which CM can generate code, provided the code generation routines are

suitably modified. There are a number of these machines for which this modification will be very straightforward. The Intel 8080 is the machine for which the modification would require the least effort. The 6502 is another. The re-writing of code generating routines has already been started with the PASCAL P-machine as the target.

LIST OF REFERENCES

LIST OF REFERENCES

(1) Aho, A.V., and Ullman, J.D., The Theory of Parsing, Translation and Compiling, Volume I: Parsing. Englewood Cliffs, NJ: Prentice-Hall, 1972.

(2) Aho, A.V. and Ullman, J.D., The Theory of Parsing, Translation and Compiling, Volume II: Compiling. Englewood Cliffs, NJ: Prentice-Hall, 1973.

(3) Aho, A.V., and Ullman, J.D., Principles of Compiler Design. Reading, MA: Addison-Wesley, 1977.

(4) Cleveland, J.C., and Uzgalis, R.C., Grammars for Programming Languages. New York: Elsevier, 1977.

(5) Halstead, M.H., A laboratory manual for compiler and operating system implementation. New York: American Elsevier, 1974.

(6) Hecht, M.S., Data Flow Analysis of Computer Programs. New York: American Elsevier, 1977.

(7) Jensen, K., and Wirth, N., PASCAL User Manual and Report. New York: Springer-Verlag, 1974.

(8) Kernighan, B.W., and Ritchie, D.M., The C Programming Language. Englewood Cliffs,NJ: Prentice-Hall, 1978.

(9) Lewis, P.M., Rosenkrantz, D.J., and Stearns, R.E., Compiler Design Theory. Reading, MA: Addison-Wesley, 1976.

(10) Snyder, A., A portable compiler for the language C. Cambridge: Massachusetts Institute of Technology Project MAC, 1975.

(11) Wulf, A.W., The Design of an Optimizing Compiler. New York: American Elsevier, 1975.

(12) Applied Computation Theory; edited by Yeh, R.T. Englewood Cliffs, NJ: Prentice-Hall, 1976.

(13) Compiler Construction: an advanced course; edited by Bauer, F.L. and Eickel, J. Berlin, West Germany: Springer-Verlag, 1974.

(14) Currents in the Theory of Computing; edited by Aho, A.V. Englewood Cliffs, NJ: Prentice-Hall, 1973.

(15) Burkowski, Forbes J., personal communication with the author.

VITA

Joseph F. Hull was born on September 14, 1948, in Quincy, Florida. He graduated from Marianna High School, Marianna, Florida, in June 1966. He attended Chipola Junior College in Marianna, Florida from September, 1966 until May, 1968 when he enlisted in the United States Air Force.

After a four-year tour of duty as a programmer at Second Air Force Headquarters, Barksdale Air Force Base, Shreveport, Louisiana, he enrolled at the Georgia Institute of Technology, Atlanta, Georgia. He received a Bachelor of Science in Applied Mathematics from that institution in May, 1974.

On graduation, Mr. Hull accepted a job at Oak Ridge National Laboratory, working on the RECON Information Retrieval System. While at ORNL, he also assisted in the design and programming of a disk access method to replace IBM's ISAM and consulted for the Regional and Urban Studies Technical Information Center(RUSTIC).

In late 1977, Mr. Hull decided to return to academia for an advanced degree and in January, 1978, he enrolled in the Graduate School of the University of Tennessee, Knoxville, Tennessee. In August, 1980, he received the Master of Science degree with a major in Computer Science and minors in Electrical Engineering and Education.

Mr. Hull is a member of the Association for Computing Machinery and the Knoxville Power Squadron. He is a volunteer worker for the Knoxville Rape Crisis Center.