

Code generation and execution of a hypothetical machine
by Per Erik Gullberg

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University
© Copyright by Per Erik Gullberg (1995)

Abstract:
This thesis discusses a high level approach for code generation and execution of a hypothetical machine. The machine consists of a set of high level instructions that reflect language concepts rather than a traditional set of low level instructions reflecting machine architecture.

A compiler for a subset of the ANCI C language was developed using ANSI C and the Purdue Compiler Construction Tool Set (PCCTS) parser development tool. A library for creating and manipulating three-dimensional vector based graphics has been added as an illustration of the modular capabilities of the compiler.

Code Generation and Execution of a Hypothetical Machine

by

Per Erik Gullberg

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

Montana State University
Bozeman, Montana

December 1995

N378
G944

APPROVAL

of a thesis submitted by

Per Erik Gullberg

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

12/15/95
Date

Chairperson, Graduate Committee

Approved for the Major Department

12/15/95
Date

Head, Major Department

Approved for the College of Graduate Studies

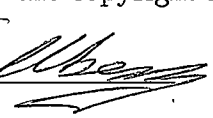
12/21/95
Date

Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature 

Date 12/20/95

Contents

Table of Contents iv

List of Tables vi

List of Figures vii

Abstract ix

1 Introduction 1
- 1.1 Overview . . . 1
- 1.2 Previous Work . . . 2
- 1.3 Preview . . . 3

2 Hypothetical Machine 4
- 2.1 Overview . . . 4
- 2.2 Code Execution . . . 5

3 C-Subset Language 7
- 3.1 Overview . . . 7
- 3.2 Variables . . . 8
- 3.3 Constants . . . 9
- 3.4 Operators . . . 9
- 3.5 Control Flow . . . 9
- 3.6 Functions . . . 10
- 3.7 Pointers & Arrays . . . 10
- 3.8 Structures . . . 12

4 PCCTS Compiler Tool 14
- 4.1 Overview . . . 14
- 4.2 Tokens . . . 15
- 4.3 Grammar . . . 16

5 C-Subset Compiler 18
5.1 Overview . . . 18
5.2 Symbol Table . . . 18
5.2.1 Structure . . . 18
5.2.2 Support Functions . . . 20
5.3 Code Generation . . . 20
5.4 Instruction Set . . . 25
5.5 Functions . . . 25
5.6 Memory Management . . . 27

6 Libraries 29
6.1 Overview . . . 29
6.2 3-D Graphics Library . . . 30

Bibliography 33

Appendix C-Subset Grammar 34

List of Tables

3.1 Precedence and Associativity of Operators in C-Subset 10

List of Figures

1.1 Typical Machine Architecture 2

2.1 Hypothetical Machine 4
2.2 Hypothetical Machine in Functional Form 5
2.3 Recursive Hypothetical Machine 5
2.4 Linked List Instruction Structure 6
2.5 Linked List Argument Structure 6
2.6 Internal Workings of Hypothetical Machine 6

3.1 C-Subset Program Illustrating Implicit Variable Declaration . . 8
3.2 Local Variable in ANSI C 9
3.3 Multi-dimensional Array Example in C-Subset 11
3.4 Structure Example in C-Subset 13

4.1 Bulding a Scanner & Parser with PCCTS 15
4.2 Multiple Automatons with PCCTS 16
4.3 Grammar Extract with Actions 17
4.4 Grammar Extract without Actions 17

5.1 Linked List Table Structure 19
5.2 Linked List Table Entry Structure 19
5.3 While-Loop Construct 21
5.4 Assembly Language Code Generated for While-Loop Construct in Figure 5.3 21
5.5 While-Loop Instruction for C-Subset 22
5.6 Code Segment Element Structure 22
5.7 Code Segment Stack 22
5.8 Code Segment Stack During While-Loop Compilation 23
5.9 Grammar Fragment Without Actions From c.g Illustrating While-Loop 24
5.10 Grammar Fragment With Actions From c.g Illustrating While-Loop 24
5.11 Support Function to Add the whileInstruction 26
5.12 Function Structure 26

6.1 Function to be Used for Adding User Defined Functions to C-Subset 30
6.2 Template For User Defined Functions That Can Be Called Form C-Subset 30
6.3 User Defined Function Argument Structure 31

Abstract

This thesis discusses a high level approach for code generation and execution of a hypothetical machine. The machine consists of a set of high level instructions that reflect language concepts rather than a traditional set of low level instructions reflecting machine architecture.

A compiler for a subset of the ANCI C language was developed using ANSI C and the Purdue Compiler Construction Tool Set (PCCTS) parser development tool. A library for creating and manipulating three-dimensional vector based graphics has been added as an illustration of the modular capabilities of the compiler.

Chapter 1

Introduction

1.1 Overview

Translating a high level program, such as C or Pascal, into machine language requires an understanding of the target machine architecture. Furthermore, it also requires an understanding of the operating system running on top of the architecture. This thesis deals with setting up a hypothetical machine which is freed from any architectural or operating system dependent issues for building a compiler.

This thesis discusses a high level approach for code generation and execution of a hypothetical machine. The machine consists of a set of high level instructions that reflect language concepts rather than a traditional set of low level instructions reflecting machine architecture.

A compiler for a subset of the ANCI C language was developed using ANSI C and the Purdue Compiler Construction Tool Set (PCCTS) parser development tool. Some reasons for using one high level language to construct another include:

- incorporating a compiler into an application for executing macros, scripts, user defined functions, or sub-programs
- constructing a portable compiler

- computer modeling for language design
- computer modeling for machine architectural design appropriate for a specific language

1.2 Previous Work

Building a computer program consists of issuing a set of instructions to be executed by a machine. The machine can either exist as hardware or software. If the machine exists only in software, then an interpreter will have to be created to emulate the machine described in software on the hardware that the software resides on. Whether the machine exists as actual hardware or as software, a typical machine can be described as seen in Figure 1.1.

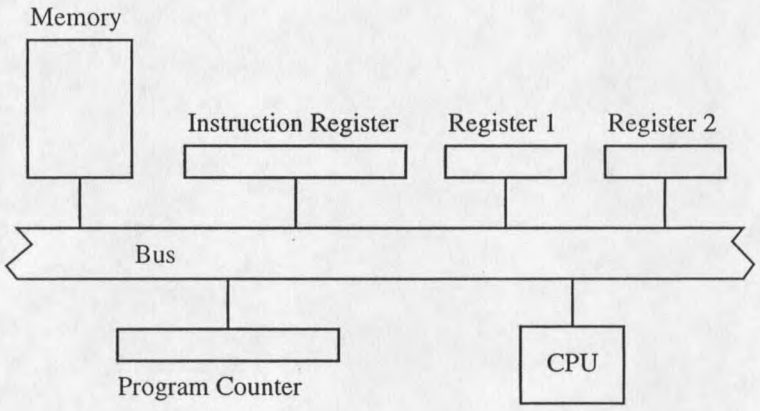


Figure 1.1: Typical Machine Architecture

The read-execute cycle of this architecture is modeled as:

1. Read the next instruction from memory and load it into the instruction register.
2. Increment the program counter.
3. Decode the instruction by setting up the registers.
4. Execute the instruction.
5. Continue from step 1 with the next instruction.

Elaborating on Figure 1.1 is easy to do if we can describe the machine as a virtual machine that is designed with software. But the underlying hardware of any machine does typically not diverge much from Figure 1.1.

A compiler is responsible for translating a computer language into a set of instructions for a machine. Usually the compiler converts a computer language to assembly code and then an assembler is responsible for converting the assembly code to machine language instructions. This is typically the way a C, Ada, or FORTRAN program is created. Other languages, such as FORTH, build a program by starting out with a set of primitive library routines and then building programs based on combining the library routines. Some languages such as BASIC and scripting languages generate code during the execution of the program.

1.3 Preview

This thesis uses the theme similar to a typical C, Ada, or FORTRAN compiler. However, instead of creating an assembly language file, code is generated directly in computer memory as linked instruction lists. These linked instruction lists are created with the aid of a code segment stack. The use of a code segment stack for generating an executable program is the distinguishing aspect of this thesis.

Chapter 2

Hypothetical Machine

2.1 Overview

Figure 2.1 shows a machine representing a hypothetical model for all computers. This model takes the black box approach for executing a program.

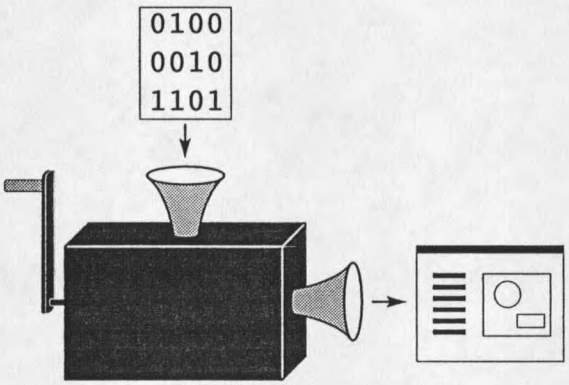


Figure 2.1: Hypothetical Machine

This black boxed machine takes as its input a compiled program, and after turning the box's handle executes the program. For the compiler that this thesis discusses, this is all that we need to know about the compiler's target machine architecture. This black boxed approach can be rewritten in functional form as shown in Figure 2.2, where the function `execute` is the black box, and `code` is the computer program fed to the machine.

```
execute( code )
```

Figure 2.2: Hypothetical Machine in Functional Form

The machine represented in Figure 2.2 can easily be created with any computer language that supports functions. To simplify and divide programming tasks another feature of the machine in Figure 2.2 is that it has recursive calling capabilities. That is to say the machine can execute a program as illustrated in Figure 2.3.

```
execute( code ) {
    .
    .
    execute( subCode );
    .
    .
}
```

Figure 2.3: Recursive Hypothetical Machine

Since this is a hypothetical machine which is implemented with a recursive-functional high level language, the compiler developer has at his disposal the features available in the high level language for creating an appropriate instruction set for compiling and executing a computer program written in a different language.

2.2 Code Execution

The machine presented in Figure 2.2, takes as its input code, which will be represented as a pointer to a linked list of program instructions. The input argument code is defined as the structure shown in Figure 2.4.

```
struct codeStruct {
    int                     (*instruction)(struct argument *);
    struct argumentStruct   *argPtr;
    struct codeStruct       *next;
};
```

Figure 2.4: Linked List Instruction Structure

Each instruction executed will be passed an argument list, which has the structure defined in Figure 2.5.

```
struct argumentStruct {
    void                    *ptr;
    struct argumentStruct   *next;
};
```

Figure 2.5: Linked List Argument Structure

Now the program code passed to machine execute() will be executed as shown in Figure 2.6. The program will halt once the end of the linked instruction list has been reached.

```
execute( struct codeStruct *code ) {
    while( code != NULL ) {
        code->instruction( code->argPtr );
        code = code->next;
    }
}
```

Figure 2.6: Internal Workings of Hypothetical Machine

Chapter 3

C-Subset Language

3.1 Overview

A compiler for a subset of the ANSI C language will be presented. Through out the remainder of this thesis the language being presented will be referred to as C-Subset. The generated code that the compiler produces from a C-Subset program will execute when given to the hypothetical machine described in chapter 2.

The grammar used for C-Subset is taken from [Kernighan 88] which had been modified by [Eneboe 95] to work with the PCCTS parser tools. Further modifications were then made to the grammar to reflect C-Subset. The complete grammar is given in the Appendix.

Features that are part of ANSI C, but not part of C-Subset are mainly:

- no variable type declaration
- no structures
- no preprocessor derivatives

For detailed information on the features available in C-Subset [Kernighan 88] can be referenced.

3.2 Variables

Since C-Subset does not declare variables, variables are declared implicitly. Figure 3.1 shows a C-Subset program where x is implicitly declared. The internal representation of all variables is a decimal number. While this model for representing variables consumes more memory that necessary at times, it greatly simplifies the programming and compilation cycle. No type specification, type casting, or type converting needs to be done by the programmer or the compiler. Furthermore since the size of all variables is transparent to the programmer, there is no need for the `sizeof` operator, which is part of ANCI C.

```
main( )
{
    x = 3 * 2.1;
    printf("x = %g\n", x);
}
```

Figure 3.1: C-Subset Program Illustrating Implicit Variable Declaration

ANSI C allows for variables to be declared with different scope levels as seen in Figure 3.2. Redeclaration of a variable within a set of braces constitutes that the variable is local with respect to the enclosing braces. Since C-Subset does not allow for variables to be declared, local redeclaration of variables is not possible. Only two scope levels exist in C-Subset, namely global and functional.

```
main( )
{
    int x;
    .
    .
    {
        int x;
        .
        .
    }
    .
    .
}
```

Figure 3.2: Local Variable in ANSI C

3.3 Constants

C-Subset does not distinguish between integers or floating point constants. All numbers, characters, and pointers are represented internally as a decimal number. The enum specifier is also supported by C-Subset, and will prove to be very useful for creating structures as arrays.

3.4 Operators

All ANSI C operators except the ones dealing with structures, type casting, and size are defined in C-Subset. Table 3.1 shows the precedence and associativity of the operators available in C-Subset.

3.5 Control Flow

The control flow statements available in C-Subset include *if*, *if-else*, *for*, *while*, *do-while*, *switch*, and *break*. Flow control statements that are part of ANSI C

Operators	*Associativity*
( ) [ ]	left to right
! ~ ++ -- + - * &	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	left to right
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Table 3.1: Precedence and Associativity of Operators in C-Subset

but not part of C-Subset are *continue*, *goto*, and labels.

3.6 Functions

Just like variables, neither the function itself nor the function's arguments are declared with a type specifier as in ANSI C. All functions return a value whether specified or not with a `return` statement. If no `return` statement is present within a function, the function will just return a zero.

3.7 Pointers & Arrays

A pointer is a variable containing the address location of a variable or a dynamically allocated block of memory. Pointers in C-Subset work the same as in ANSI C, however pointer arithmetic is only done via array indexing in C-Subset. That is to say in ANSI C if the variable x is defined as a pointer to 10 integer numbers, then the third element of x could be accessed as:

1. *(x+2)

or

2. x[2]

Access method number 1 is not possible with C-Subset because it cannot determine whether x+2 is to perform pointer or numerical arithmetic since variables are not declared. However, the array indexing syntax of access method number 2 suggests that x is a pointer.

Figure 3.3 is a C-Subset program demonstrating how a two-dimensional array could be created. The function make2D is a function that will create a two-dimensional array containing row rows and col columns. In main, array gets assigned the starting address location of the allocated two-dimensional array created by make2D. Then the first column element of the second row of array is set to 921. This example could be extended to any number of dimensions.

```
make2D(row, col)
{
    ptr = malloc(row);              /* allocate 'row' pointer elements */

    for(i=0; i<row; i++)            /* allocate 'col' elements for each row */
       ptr[i] = malloc(col);

    return(ptr);                    /* return pointer of new 2D array */
}

main()
{
    array = make2D(2, 3);           /* create a 2 X 3 array */

    array[1][0] = 921;              /* set the first column element
                                       of the second row to 921 */
}
```

Figure 3.3: Multi-dimensional Array Example in C-Subset

3.8 Structures

Even though structures are not part of C-Subset, a structure can still be created with arrays. This can be done due to all variables having the same size. Since enum starts numbering by default from zero, and then increments by one for each successive enumerated constant, the defined constants can be used to access various elements of an array with a labeled constant. Figure 3.4 shows an example where the enum specifier is used to declare structures in C-Subset. The example explains how a structure for a two-dimensional line could be set up. Line 1 defines a structure of a two-dimensional coordinate, and line 7 defines the structure of a two-dimensional line. The elements `CORD_SIZE` and `LINE_SIZE` are declared for automatic calculation of the structure sizes, and will be handy during the memory allocation process of these structures. Line 26 calls the function `createLine` to do the necessary memory allocation for creating a new line. Line 16 allocates the line stucture, and lines 18 and 19 allocate the coordinate structure of the `LINE_START` and `LINE_END` elements of the new line. Once `line` at line 26 has been created, the structure fields of `line` can be accessed as seen in lines 28 through 34.

```
1   enum {          /* structure declaration of 2D coordinate */
2       CORD_X,
3       CORD_Y,
4       CORD_SIZE
5   };
6
7   enum {          /* structure declaration of 2D line */
8       LINE_START,
9       LINE_END,
10      LINE_COLOR,
11      LINE_SIZE
12  };
13
14  createLine()
15  {
16      newLine = malloc(LINE_SIZE);              /* allocate line structure */
17
18      newLine[LINE_START] = malloc(CORD_SIZE);  /* allocate cord structure */
19      newLine[LINE_END] = malloc(CORD_SIZE);    /* allocate cord structure */
20
21      return(newLine)    /* return pointer to new line */;
22  }
23
24  main()
25  {
26      line = createLine();          /* create a line */
27
28      line[LINE_START][CORD_X] = 0;  /* set start coordinate to (0,0) */
29      line[LINE_START][CORD_Y] = 0;
30
31      line[LINE_END][CORD_X] = 1     /* set end coordinate to (1,2) */;
32      line[LINE_END][CORD_Y] = 2;
33
34      line[LINE_COLOR] = 4;          /* set color to color #4 */
35  }
```

Figure 3.4: Structure Example in C-Subset

Chapter 4

PCCTS Compiler Tool

4.1 Overview

The scanner and parser for C-Subset originated from the LEX and YACC, but for reason discussed in section 4.3 switched to the Purdue Compiler Construction Tool Set (PCCTS). PCCTS is a set of public domain software tools for developing compilers and other translation systems [Parr 93]. Two programs of the PCCTS tool set were used for developing C-Subset, namely ANTLR (ANother Tool for Language Recognition) and DLG (Determanistic finite automaton - based Lexical analyzer Generator). Some of the key files that ANTLER and DLG use and generate are:

c.g

Contains C-Subset token and grammer information, and is the input file for ANTLR.

c.c

ANSI C parser produced by ANTLR via c.g.

parser.dlg

Scanner specification for DLG produced by ANTLR via c.g.

scan.c

ANSI C scanner produced by DLG via parser.dlg.

Figure 4.1 shows how the scanner and parser was generated with ANTLER and DLG.

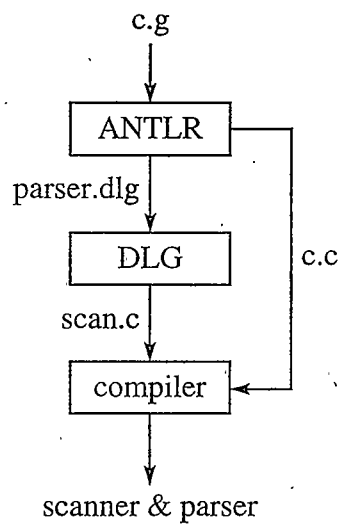


Figure 4.1: Bulding a Scanner & Parser with PCCTS

4.2 Tokens

In regards to the input file c.g, the token definitions represent the terminal symbols of the defined grammar. Regular expressions are used for defining tokens. PCCTS allows optional actions to be associated with any token. That is to say, when the scanner encounters some token, action code can be executed. Action code is entered between << >> as seen in Figure 4.2. Figure 4.2 also illustrates the use of multiple automatons for declaring tokens. Multiple automatons are used to recognize conflicting regular expressions within the same lexical analyzer, such as comments. In Figure 4.2, upon entry of a comment (/*), the action in line 2 calls the `COMMENT` automaton. The `COMMENT`

automaton then takes over until the end of the comment (*/) is found. In line 6, control is returned back to the default START automaton.

```
1  /* lexclass START is in effect by default */
2  #token "/\*"          << zzmode(COMMENT);  /* switch to COMMENT automaton */
                            zzskip (); >>
3   ...
4  #lexclass COMMENT
5  #token "[\n\r]"       << zzline++; zzskip(); >>
6  #token "\*/"          << zzmode(START);    /* switch back to START automaton */
                            zzskip(); >>
7  #token "\*"           << zzskip(); >>
8  #token "~[\*\n\r]+"   << zzskip(); >>
9   ...
```

Figure 4.2: Multiple Automatons with PCCTS

4.3 Grammar

Grammar rules employed by PCCTS use extended BNF constructs. PCCTS produces LL(k) parsers, while YACC produces LALR(1) parsers. LL parsers builds the syntax tree top-down, by starting with a single non-terminal at the top of the tree and working its way down to the terminals. LALR builds the syntax tree bottom-up, starting from the terminals at the bottom of the syntax tree and working its way up.

Building the syntax tree in a top-down fashion allows for actions to be embedded anywhere within the grammar rules, while building the syntax tree in a bottom-up fashion allows for actions to be placed only at the end of each grammar rule. The reason for this is because top-down knows the production that will expand the rule's left-hand side *before* the right-hand side is parsed, by matching the pre-calculated predict sets to the lookahead symbols. Whereas bottom-up has to first parse the right-hand side before the left-hand side can be deduced.

Since the compiler for C-Subset is a one-pass compiler and code is being

generated directly, it is often convenient to place actions preceding certain production segments (i.e. embedded actions). For this reason the PCCTS tools turned out to be a better choice than using YACC. Figure 4.3 is an extract from c.g which shows heavy usage of actions (code between << >>) embedded within the grammar.

```
selection_statement :

    << int hasElse = FALSE; >>

    IF LPAREN << c_pushCodeSegment(); >> expression_value RPAREN
    << c_pushCodeSegment(); >> statement

    { ( ELSE )? ELSE << c_pushCodeSegment(); >> statement << hasElse = TRUE; >> }

    << if(hasElse) c_addExecuteIfThenElse();
       else c_addExecuteIfThen(); >>
```

Figure 4.3: Grammar Extract with Actions

Rewriting Figure 4.3 without the actions makes the actual grammar easier to read as seen in Figure 4.4. Figure 4.4 shows how the classic *if-then-else* statement could be written with PCCTS. Line 3 shows the *if-then* clause, and line 5 shows the *else* clause. The *else* clause is declared as optional by placing it within curly braces. PCCTS allows for infinite syntactic predicates to aid in resolving ambiguous situations. A syntactic predicate is written in PCCTS as (*declaration*)? where *declaration* must be satisfied upon proceeding with a production. Without the syntactic predicate (ELSE)? in line 5, the rule would be ambiguous.

```
1  selection_statement :
2
3      IF LPAREN expression_value RPAREN statement
4
5      { ( ELSE )? ELSE statement }
```

Figure 4.4: Grammar Extract without Actions

Chapter 5

C-Subset Compiler

5.1 Overview

The C-Subset compiler consists of several parts. First an explanation of the duties of the symbol table is given. The method of code generation for the hypothetical machine described in chapter 2 is then discussed and is considered the heart of this thesis. Other issues relevant to the internal workings of the compiler are then discussed. These issues include such things as the instruction set, functions, and memory management.

5.2 Symbol Table

5.2.1 Structure

The symbol table is used for storing and retrieving information about various symbols found in a computer program during the compilation process. Typically the symbol table holds information on the program's variables. Information can include such things as name, type, and scope.

For C-Subset the symbol table will be represented with a linked list having the structure shown in Figure 5.1. Each time a new scope level is encountered in the program being compiled, a new symbol table entry is added to the front

of the linked list, and acts as the current symbol table.

```
struct table {
    struct tableEntry   *entry;
    int                 snl;
    int                 tableSize;
    struct tableEntry   **currentEntry;
    struct table        *next;
};
```

Figure 5.1: Linked List Table Structure

The structure entries of Figure 5.1 are:

- entry - pointer to the table entries
- snl - scope nesting level of the table
- tableSize - number of entries in the table
- currentEntry - pointer to where the next table entry will be added
- next - pointer to the next table

The entry field in Figure 5.1 is defined as the structure defined in Figure 5.2. Since C-Subset has no variable declarations, only the variable name and an indication as to if the variable is part of a function argument list will be necessary for parsing C-Subset.

```
struct tableEntry {
    char                *name;
    char                isArgument;
    struct tableEntry   *next;
};
```

Figure 5.2: Linked List Table Entry Structure

The structure entries of Figure 5.2 are:

- name - name of the variable as defined in the program
- isArgument - variable to aid in the compilation process
- next - pointer to the next entry

5.2.2 Support Functions

During the compilation phase, the compiler needs to work with the symbol table. The following set of symbol table support functions exists.

pushTable
Pushes another table onto the symbol table stack. This is done each time the compiler enters a new scope level.

popTable
Pops the symbol table stack. This is done each time the compiler exits a scope level.

getTopTableSize
Gets the number of symbol entries from the top most table of the symbol table stack.

pushEntry
Adds a new symbol entry to the top most table of the symbol table stack.

getEntry
Gets information on the requested symbol.

5.3 Code Generation

A common way to generate code for the while-loop construct seen in Figure 5.3 is to create two unique labels, say `Label_1` and `Label_2`, upon realizing the token `while`. Then to use those two labels to set up assembly type language code as seen in Figure 5.4.

A slightly different approach will be used for the compiler explained in this thesis. Referring back to Figure 2.6 on page 6, all the hypothetical machine can do is execute a list of instructions until the end of the list is reached.

```
while( exp )
    statement
```

Figure 5.3: While-Loop Construct

```
.
.
Label_1:
  - code for exp
  - jump to Label_2 if result of exp equals 0
  - code for statement
  - jump to Label_1
Label_2:
.
.
```

Figure 5.4: Assembly Language Code Generated for While-Loop Construct in Figure 5.3

With this in mind and the fact that the machine can be called recursively, the instruction for handling the while-loop construct in Figure 5.3 could be written in C as shown in Figure 5.5. The way the code segment lists `expCode` and `statementCode` are created during the compilation phase is with the help of a code segment stack.

Figure 5.6 shows the structure of a code segment stack element. Figure 5.7 shows what the code segment stack might look like at some point during the compilation process. In Figure 5.7, `topCodeSegment` always points to the top element of the stack, and `currentCode` always points to where the next instruction will be added. When the stack is popped `currentCode` will be set to `nextNextInst` of the element being popped off.

```
whileInstruction(expCode, statementCode)
{
    execute(expCode);              /* execute exp */
    while(popStack() != 0) {       /* check result of exp */
        execute(statementCode); /* execute statement */
        execute(expCode);          /* execute exp */
    }
}
```

Figure 5.5: While-Loop Instruction for C-Subset

```
struct codeSegment {
  struct codeSegment *next;          /* link to next
                                        code segment */
  struct code        **nextNextInst; /* where to pick up once
                                        the stack is poped */
  struct code        **codeSeg;      /* pointer to beginning
                                        of code segment */
};
```

Figure 5.6: Code Segment Element Structure

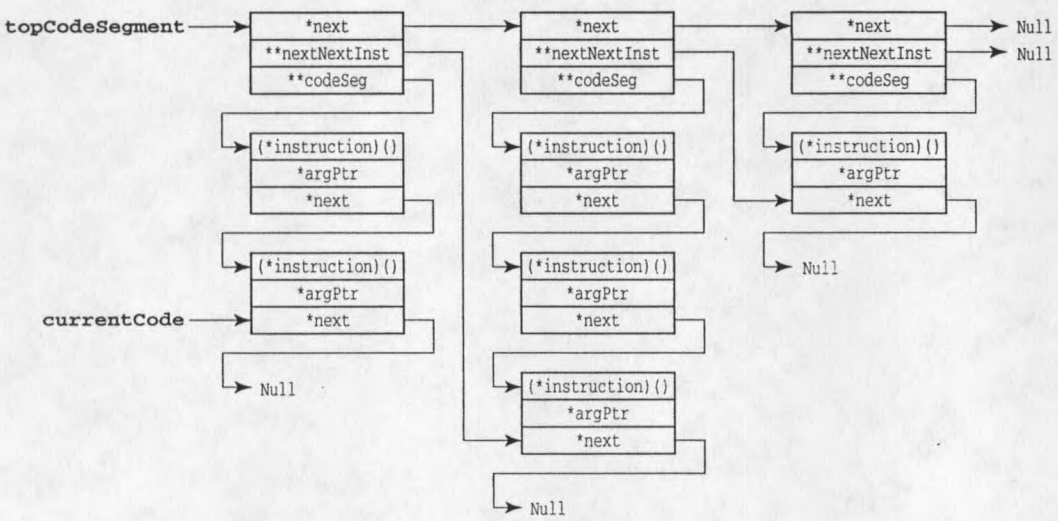


Figure 5.7: Code Segment Stack

The grammar for the while-loop construct of C-Subset is as shown in Figure 5.9. In reference to Figure 5.8, if code is currently being added to `mainCode`, then before *exp*, in Figure 5.9, is expanded upon, a new code segment will be pushed onto the code segment stack, as seen in line 2 of Figure 5.10. This new code segment will contain only those instructions relating to *exp*. Likewise a separate code segment will be created for *statement* in Figure 5.9. Once the last instruction for *statement* has been added, the action function `addWhileInstruction()` is called as seen in line 4 of Figure 5.10.

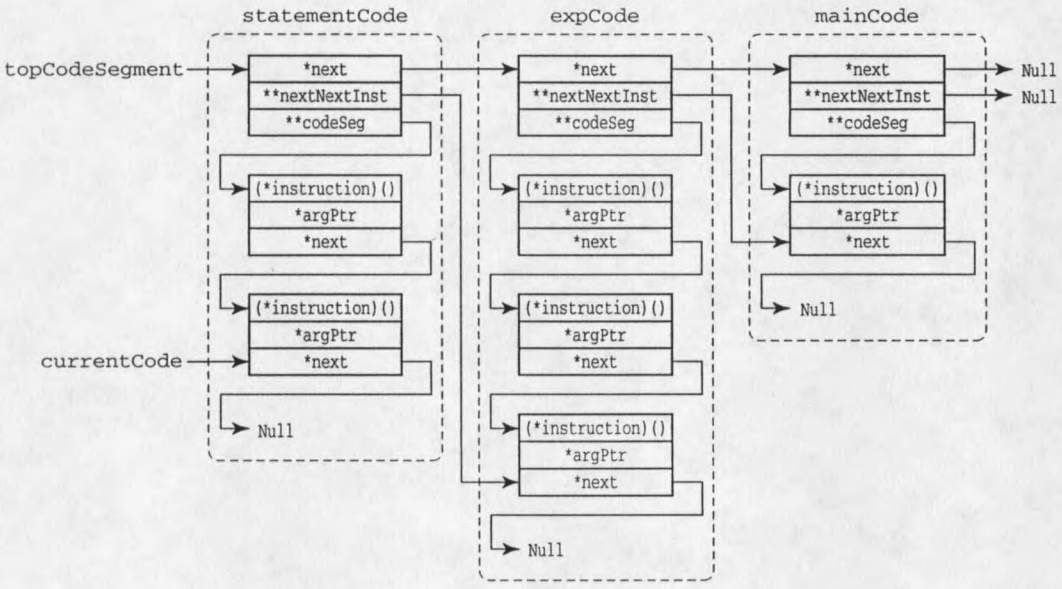


Figure 5.8: Code Segment Stack During While-Loop Compilation

`addWhileInstruction()` will be responsible for adding the

```
whileInstruction(expCode, statementCode)
```

instruction, seen in Figure 5.5, to the end of the code segment `mainCode`, which is currently the third from the top stack element as seen in Figure 5.8. This is done by:

```
iteration_statement:
    WHILE LRAREN exp RPAREN
        statement
    ;
```

Figure 5.9: Grammar Fragment Without Actions
From c.g Illustrating While-Loop

```
1  iteration_statement:
2      WHILE LRAREN << pushCodeSegment(); >> exp RPAREN
3          << pushCodeSegment(); >> statement
4          << addWhileInstruction(); >>
5      ;
```

Figure 5.10: Grammar Fragment With Actions
From c.g Illustrating While-Loop

1. Saving the starting location of code segment *statement* into a temporary variable called `statementCode`.

2. Popping the code segment stack.

3. Saving the starting location of code segment *exp* into a temporary variable called `expCode`.

4. Popping the code segment stack.

The instruction `executeWhile(expCode, statementCode)` can now be added to `mainCode`, and it will operate as was shown in Figure 5.5.

5.4 Instruction Set

The instructions available for C-Subset are declared and defined in the files `inst.h` and `inst.c`. Each time an instruction is to be added during the compilation phase, the instructions support function add_*instruction* is used. These support functions are declared and defined in `addInst.h` and `addInst.c`. The support functions add_*instruction* are used to add an instruction to the top most code segment of the code segment stack which is pointed to by `currentCode` as seen in Figure 5.7.

As discussed in Figure 5.10, line 4 adds the `whileInstruction` to the `mainCode` segment shown in Figure 5.8 via `addWhileInstruction()`. The `addWhileInstruction()` support function is shown in Figure 5.11. Lines 3 and 4 declare pointers that will hold the starting points of `expCode` and `statementCode` seen in Figure 5.8. Line 6 assigns `statementCode`. Line 7 pops the code segment stack. Line 8 assigns `expCode`. Line 9 pops the code segment stack once again. The `whileInstruction` is then added to the end of `mainCode` in line 11. Lines 12 and 13 add the arguments that `whileInstruction` will use during runtime. Finally line 14 prepares for the next instruction to be added.

5.5 Functions

Functions declared internal to program code and functions that are declared in external libraries are maintained by a central data structure as seen in Figure 5.12.

The structure entries of Figure 5.12 are:

- name - name of the function
- type - INTERNAL_FUNCTION or LIB_FUNCTION

```
 1   c_addExecuteWhile(void)
 2   {
 3       struct code   *expCode;
 4       struct code   *statementCode;
 5
 6       statementCode = *(topCodeSegment->codeSeg);
 7       c_popCodeSegment();
 8       expCode = *(topCodeSegment->codeSeg);
 9       c_popCodeSegment();
10
11       setInstruction(whileInstruction);
12       addArgument(ARG_PTR, expCode);
13       addArgument(ARG_PTR, statementCode);
14       nextInstruction();
15   }
```

Figure 5.11: Support Function to Add the whileInstruction

```
struct func {
    char          *name;
    int           type;
    int           argNum;
    struct code   **internalCode;
    int           (*externalCode)(struct argument *);
    void          *clientData;
};
```

Figure 5.12: Function Structure

- argNum - number of arguments required
- internalCode - pointer to internal code if type = INTERNAL_FUNCTION
- externalCode - pointer to external code if type = LIB_FUNCTION
- clientData - pointer to client data

During the compilation phase an array variable called `function` of type `struct func` is created. When a new function is being added, `addFunction()` is used for adding a function to `function`. Each time a reference to a function is made, `index` in `function[index]` is determined for the requested function via `getFunction()`. The index value returned by `getFunction()` is then used during runtime to access the function of interest.

5.6 Memory Management

One of the features of C-Subset is that it can be embedded within other C/C++ projects. Since C-Subset can work with pointers, a memory scheme for protecting application memory is employed. The goal of C-Subset is to create a safe environment with memory overwrite address checking.

The library `libsafeMem.a` defines a set of routines for allocating a block of memory to be used for dynamic memory allocation. These routines are only used internally at runtime of a C-Subset program. The library consists of the following routines:

`safeMem *safeMem_construct(long size)`
: Creates and initialized a `safeMem` structure. Initialization includes allocating a block of memory, of size `size`, which will be used for dynamic memory allocation during execution of a C-Subset program. A pointer to the `safeMem` structure is returned.

`void safeMem_destruct(safeMem *m)`
: Disposes of the `safeMem` structure `m`.

`void *safeMem_malloc(safeMem *m, long request)`
Returns a pointer to dynamically allocated memory from `m`.

`int safeMem_free(safeMem *m, void *freePtr)`
Frees `freePtr` from `m`.

`int safeMem_check(safeMem *m, long address)`
Checks if address is within legal bounds.

Chapter 6

Libraries

6.1 Overview

So far only a fraction of the ANSI C libraries have been implemented for C-Subset. Library files start with the letter "l". The standard library functions supported in the relevant files are:

lStdio.c
printf, sprintf, scanf

lStdlib.c
malloc, system

lString.c
strcpy, strlen

lMath.c
sin, cos, atan, abs

To add user defined libraries the support function seen in Figure 6.1 can be used. The arguments for the function are:

- name - name of the function
- function - pointer to the function
- clientData - pointer to user data

```
int c_addFunction(char *name, C_func function, void *clientData);
```

Figure 6.1: Function to be Used for Adding User Defined Functions to C-Subset

The function pointer passed to `c_addFunction` is of type `C_func` which has the form as seen in Figure 6.2. The argument for `template` is of type `C_funcInfo` which is the structure seen in Figure 6.3. The structure entries of Figure 6.3 are:

- argc - number of arguments passed to function
- argv - array holding argument values
- clientData - pointer to user data defined via `c_addFunction`
- codeLineNo - line number in C-Subset program where function call resides
- result - function result

```
int template(C_funcInfo *info)
{
    /* user defined code */
}
```

Figure 6.2: Template For User Defined Functions That Can Be Called Form C-Subset

6.2 3-D Graphics Library

Part of the inspiration for doing this thesis was to create a programming environment for manipulating three-dimensional graphics objects. Work prior to

```
typedef struct {
    int     argc;
    double  *argv;
    void    *clientData;
    int     codeLineNo;
    double  result;
} C_funcInfo;
```

Figure 6.3: User Defined Function Argument Structure

this thesis consisted of setting up a data base for graphics to be used in conjunction with a set of graphics routines. A library of these graphics routines, for use with C-Subset, was then created. This three-dimensional graphics library is defined in the file `l3D.C`, and can serve as a template for future user defined libraries. A brief description of the library's functions are given below.

`newWindow3D()`
: Returns a pointer to a new three-dimensional graphics window.

`newObject3D(win, parent)`
: Adds a new object to `win`. If the new object is created relative to `parent`, then any matrix transformations that are applied to `parent` are also applied to the new object. Passing `NULL` for `parent` indicated that the new object should be a root object. Relative objects can be created indefinitely, and is a powerful way for doing complex transformations on objects. A pointer to the new object is returned.

`newPoly3D(object)`
: Adds an empty polygon item to `object`. A pointer to the item is returned.

`addPoint3D(item, x, y, z)`
: Adds the coordinate (`x`, `y`, `z`) to `item`.

`clear3D(win)`
: Clears the window `win`.

`draw3D(win)`
: Queries `win`'s object data base to draw the contents of `win`.

`translate3D(object, x, y, z)`
Translates `object` according to the values of `x`, `y` and `z`.

`scale3D(object, x, y, z)`
Scales `object` according to the values of `x`, `y` and `z`.

`rotate3D(object, degrees, axis)`
Rotates `object` about `axis` 1, 2, or 3 (x, y, or z) `degrees` degrees.

`lineRotate3D(object, x1, y1, z1, x2, y2, z2, degrees)`
Rotates `object` about the line passing through coordinate (`x1`, `y1`, `z1`) and (`x2`, `y2`, `z2`) `degrees` degrees.

Bibliography

[Kernighan 88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall Inc., Englewood Cliffs, New Jersey. 1988.

[Parr 93] T. Parr, W. Cohen, and H. Dietz. *PCCTS Reference Manual*. On-line reference manual.

[Eneboe 95] T. J. Eneboe. *An ANSI C Compiler for the E-machine*. Master's thesis. Computer Science Department, Montana State University. June, 1995.

[Fischer 91] C. N. Fischer and R. J. LeBlanc, Jr. *Crafting a Compiler with C*. Benjamin/Cummings Publishing Company, Menlo Park, California. 1991.

[MacLennan 87] B. J. MacLennan. *Principles of Programming Languages*. Holt, Rinehart & Winston, Inc. 1987.

Appendix

C-Subset Grammar

This appendix lists the grammar rules for the C-Subset language which is accepted by the PCCTS parser tools. PCCTS is explained in chapter 4. Unlike the file `c.g` which also has the grammar for C-Subset, the version presented in this appendix is freed from the action routines for simplifying readability.

```
translation_unit :
          ( external_declaration )+ Eof
        ;

external_declaration :
          ( IDENTIFIER LPAREN { function_arg } RPAREN LBRACE )? function_definition
        | global_declaration
        ;

function_definition :
          IDENTIFIER LPAREN { function_arg } RPAREN
              LBRACE ( enum_specifier )* ( statement )* RBRACE
        ;

function_arg :
          IDENTIFIER ( COMMA IDENTIFIER )*
        ;

global_declaration :
          expression SEMICOLON
        | enum_specifier
        ;

enum_specifier :
          ENUM LBRACE enumerator_list RBRACE SEMICOLON
        ;

enumerator_list :
          enumerator ( COMMA enumerator )*
        ;

enumerator :
```

```
          IDENTIFIER { ASSIGNEQUAL constant_expression }
        ;

statement :
          expression SEMICOLON
        | SEMICOLON
        | LBRACE ( statement )* RBRACE
        | selection_statement
        | iteration_statement
        | jump_statement
        ;

selection_statement :
          IF LPAREN expression_value RPAREN statement { ELSE statement }
        | SWITCH LPAREN expression_value RPAREN
            LBRACE ( ( CASE expression_value COLON ( statement )* )
                   | ( DEFAULT COLON ( statement )* ) )* RBRACE
        ;

iteration_statement :
          WHILE LPAREN expression_value RPAREN statement
        | DO statement WHILE LPAREN expression_value RPAREN SEMICOLON
        | FOR LPAREN
            { expression_value } SEMICOLON
            { expression_value } SEMICOLON
            { expression_value } RPAREN statement
        ;

jump_statement :
          BREAK SEMICOLON
        | RETURN { expression_value } SEMICOLON
        ;

expression_value :
          expression
        ;

expression :
          assignment_expression ( COMMA assignment_expression )*
        ;

assignment_expression :
          conditional_expression
            { ( ASSIGNEQUAL
              | TIMESEQUAL
              | DIVIDEEQUAL
              | MODEQUAL
              | PLUSEQUAL
              | MINUSEQUAL
              | SHIFTLEFTEQUAL
              | SHIFTRIGHTEQUAL
              | BITWISEANDEQUAL
              | BITWISEOREQUAL
              | BITWISEXOREQUAL ) assignment_expression }
        ;

conditional_expression :
          logical_or_expression { QUESTIONMARK expression COLON conditional_expression }
        ;

constant_expression :
          conditional_expression
        ;
```

```
logical_or_expression :
          logical_and_expression ( OR logical_and_expression )*
        ;

logical_and_expression :
          inclusive_or_expression ( AND inclusive_or_expression )*
        ;

inclusive_or_expression :
          exclusive_or_expression ( BITWISEOR exclusive_or_expression )*
        ;

exclusive_or_expression :
          and_expression ( BITWISEXOR and_expression )*
        ;

and_expression :
          equality_expression ( AMPERSAND equality_expression )*
        ;

equality_expression :
          relational_expression ( ( EQL | NEQL ) relational_expression )*
        ;

relational_expression :
          shift_expression ( ( LESS | GTR | LEQL | GEQL ) shift_expression )*
        ;

shift_expression :
          additive_expression ( ( SHIFTLEFT | SHIFTRIGHT ) additive_expression )*
        ;

additive_expression :
          multiplicative_expression ( ( PLUS | MINUS ) multiplicative_expression )*
        ;

multiplicative_expression :
          power_expression ( ( STAR | DIVIDE | MOD ) power_expression )*
        ;

power_expression :
          unary_expression { POWER power_expression }
        ;

unary_expression :
          postfix_expression
        | INCREMENT postfix_expression
        | DECREMENT postfix_expression
        | ( AMPERSAND | STAR | PLUS | MINUS | BITWISENOT | NOT ) unary_expression
        ;

postfix_expression :
          primary_expression
              ( LBRACKET expression_value RBRACKET
              | INCREMENT | DECREMENT | LPAREN
                  { argument_expression_list } RPAREN )*
        ;

primary_expression :
          IDENTIFIER
        | constant
        | ASTRING
```

```
        | LPAREN expression RPAREN
        ;

argument_expression_list :
          assignment_expression ( COMMA assignment_expression )*
        ;

constant :
          OCTALINT
        | HEXADECIMALINT
        | ACHARACTER
        | DECIMALFLOAT
        ;
```

MONTANA STATE UNIVERSITY LIBRARIES
3 1762 10224155 9