THE IMPLEMENTATION OF AN INTERPRETER FOR A PROGRAMMING LANGUAGE

GRADUATE PROJECT

BY
DAVID MOSEL

COMMITTEE MEMBERS

Dr. R. Stephen Dannelly,
Chairman

Dr. Mario Guimaraes,
Member

Dr. Holly Patterson,
Member
Abstract

This project is an implementation of an interpreter for a simple programming language. The interpreter may be used by anyone who wishes to study the translation of a high-level language into three-address code. The interpreter can be used in this way because it can show the three-address code generated during parsing, show the execution of the three-address code, and show the firing actions to reduce grammar productions during parsing. The interpreter is capable of performing floating-point computations and is designed for use on single-source-file programming solutions that require assignment to variables, flow of control statements, subroutine calls, and recursion. The syntax for the programming language is a modified subset of the C programming language. The interpreter works by translating program source code into a form of three-address code known as quadruples. These quadruples are pushed onto a stack during parsing, and serve as the object code which the interpreter executes. The Unix tools Flex and YACC were used in the interpreter implementation. Flex was used to create a lexical analyzer, and YACC was used to generate a parser for the programming language.
# Table of Contents

1 Introduction and Background  
1.1 Interpreters  
1.2 This Interpreter  
1.3 This Document  

2 Description of the Interpreter  
2.1 Language Definition  
2.1.1 Operators  
2.1.2 Flow of Control Statements  
2.1.3 Subroutines  
2.1.4 Print Statements  
2.1.5 Return Statements  
2.1.6 Break Statements  
2.1.6 Numbers  
2.1.7 Variables  
2.1.8 Comment Convention  
2.2 Notes  
2.2.1 Scope Rules  
2.2.2 Invoking the Interpreter  
2.2.3 System User  
2.3 Examples  

3 Environment
4 Overview of Flex and YACC 19

5 System Design 23

5.1 Description of Project Elements 23
5.2 Procedure Overview 28
5.3 Steps to Completion of the Project 29

5.3.1 Implementation of the Desk Calculator 29
5.3.2 Modification of the Desk Calculator into a Small Interpreter for Quadruples 37
5.3.3 Addition of Variables and Variable Assignments 55
5.3.4 Addition of Flow of Control Statements 61
5.3.5 Addition of Subroutine Definitions 73
5.3.6 Addition of Subroutine Calls and Return Statements 84
5.3.7 Addition of Recursive Subroutine Calls 93
5.3.8 Addition of Switches 101

6 Results 106

7 Summary and Future Work 110

References 112

Appendix - Program Source Code 113
List of Figures

4.1. Creation of a lexical analyzer with Flex. 20
4.2. Creation of a parser with YACC. 22
5.1. Graphical representation of the system design. 23
5.2. Outline of the interpreter relationships. 24
5.3. Grammar for the desk calculator. 30
5.4. YACC grammar specification for the desk calculator. 31
5.5. Flex lexical analyzer specification. 34
5.6. Contents of the file main.c during step one. 36
5.7. Output from desk calculator during step one. 37
5.8. Definition of a quadruple. 37
5.9. Definition of a symbol. 38
5.10. Symbol table management functions. 39
5.11. Virtual machine operator codes. 40
5.12. Definition of Load function. 40
5.13. YACC grammar specification modified for quadruples. 41
5.14. The contents of y.tab.h during step two. 44
5.15. Definition of the interpreter function. 45
5.16. Definition of the rterr function. 47
5.17. Definition of DumpProg function. 47
5.18. Definition of the emit function. 48
5.19. The revised main function. 48
5.20. Output of interpreter during step two.
5.21. Updates to Flex specification reflecting additional binary operators.
5.22. Added productions to YACC grammar specification.
5.23. YACC grammar updates for statements.
5.24. Regular expression and actions required for strings.
5.25. Productions defining a print list.
5.27. Output from interpreter during step two.
5.28. Revision to Flex specification required for variables.
5.29. Revision to Flex specification for remaining operators.
5.30. YACC token definitions and rules of precedence in step three.
5.31. Productions defining an assignment statement.
5.32. Additional rules defining an expression.
5.33. Case statement defining equivalence operator in function DumpProg.
5.34. Output from interpreter during step three.
5.35. Updates to Flex specification for flow of control statements.
5.36. Grammar productions for compound statements.
5.37. Grammar productions defining flow of control statements.
5.38. Productions defining the non-terminal for_list.
5.39. Productions defining markers within the grammar.
5.40. Redefined YACC stack type.
5.41. Modified symbol definition.
5.42. The GetLpc and Load functions.

5.43. Production and semantic actions defining an if-else statement.

5.44. Contents of test program during step four.

5.45. Three-address code generated from the test program.

5.46. Updates to the interpreter function.

5.47. Test program and output from interpreter in step four.

5.48. New symbol table definition.

5.49. Definition of the NewTable function.

5.50. Updated lexical rule representing a variable name.

5.51. Listing of functions IsSub and isGlobal.

5.52. Production and semantic actions defining a subroutine declarator.

5.53. Definition of checkSub function.

5.54. Productions defining a subroutine variable declaration list.

5.55. Production and semantic actions defining a subroutine definition.

5.56. Rule defining program statements.

5.57. The main function during step five.

5.58. Output from parsing test program during step five.

5.59. Productions defining actual parameters.

5.60. Production defining a subroutine call.

5.61. Production defining a call within an expression.

5.62. Production defining a return statement.

5.63. Definition of activation records.
5.64. Definition of the calling sequence.
5.65. Statements defining the return value operator.
5.66. Statements defining the exit operator.
5.67. New definition for a symbol table node.
5.68. Updated definition of a symbol.
5.69. Modified subroutine production.
5.70. Macro substitutions for interpreter function.
5.71. Statements defining the multiplication operation.
5.72. Definition of initMain.
5.73. Definition of initStack.
5.74. Recursive program and output after step seven.
5.75. Final statements defining the multiplication operator.
6.1. Output from interpreter showing parser actions.
6.2. Syntax tree drawn from interpreter output.
1 Introduction and Background

1.1 Interpreters

Interpreters are important system tools used throughout the computing industry. They are important because they can be used as programming language interpreters or as command interpreters. They are also important because of their potential portability across platforms.

Interpreters can be used to execute programming languages. Examples of interpreted languages include LISP, ML, Prolog, Smalltalk, and BASIC. There are some programming languages that are generally interpreted only. This is because most bindings are not made until execution, or because dynamic changes in the type of an object make direct translation by a compiler into machine code extremely difficult. Lisp is an example of a language designed for research in the area of artificial intelligence that has generally only been implemented as an interpreted language.

Another important example of the use of interpreters is as command processors, or command interpreters. Several examples of command interpreters can be found in the Unix operating system. The shells Bash, Bourne-Shell, C-Shell, Korn, Tcsh, and Zsh are all examples. These shells serve as the interface between users and the operating system. Interpreters are ideal for this task because they can be designed to invoke a complex task such as an editor, compiler or system routine.
Interpreters are also important because they can have the attribute of being ported to different platforms. Two popular examples of interpreters that are being ported to different systems are Java and PERL. The reason that some interpreters have been ported to different systems is because they are generally written in a high-level language. Therefore, in order to port an interpreter written in a high-level language such as ANSI C, the source code is simply recompiled on the new system. Minor changes in the code specific to the previous platform may be required. However, porting an interpreter written in a high-level language to another platform should not involve a major rewrite of the interpreter source code.

1.2 This Interpreter

This project was selected because of my desire to learn more about and implement some of the internal workings of interpreters, command processors and compilers. This desire was further fueled by some of my course work. In Programming Languages, I had an opportunity to see some the of back end of a complicated interpreter used for instruction purposes. Additionally, in Survey of System Software, I had been given an over view of the ability of a parser generator such as YACC to implement much of the front end of an interpreter or compiler.

This project was deemed acceptable by the project chairman after instructions to include any necessary elements which would enable the interpreter to display the three-address code generated during parsing, show the execution of the three-address code during actual program execution, and show the firing actions of the parser as grammar
productions are reduced. The interpreter's ability to convey its internal function enables it to be an effective learning tool for studying and learning about the translation of a high-level language into three-address code. The tool is already in use by Dr. Dannelly in both COSC 3346 and COSC 5331.

A C-like language was selected for the language of the interpreter because of my belief that C is a language with a very simple syntax, especially when pointers are not involved. Furthermore, the C programming language has very few reserved words and few programming rules. Additionally, many people already know how to program in the C programming language or another language based on C such as PERL or Java.

Finally, this project is worthy as a graduate project because it incorporates ideas from several areas of computer science. These areas include programming languages, compiler design, system software, data structures, language theory, structured programming, systems analysis and design, and the use of existing software tools in the implementation of a complex application.

1.3 This Document

This document is about the implementation of a programming language interpreter, and how the finished interpreter can be used to study the translation of a high-level language into three-address code. It details every step required to build the simple interpreter described in the document abstract. Additionally, it is a working example
which may give insight into how programming tools such as Flex and YACC can be used to simplify a complex software project.

Chapter two is a description of the interpreter. This chapter shows how to invoke the interpreter and how the interpreter can be used to write and execute programs. The chapter describes the syntax for the programming language by giving a language definition. The language definition details all of the features of the programming language such as operators, statements, and scope rules. Additionally, several programming examples are given.

Chapter three is a concise description of the project environment. The chapter describes the environment required for building and using the interpreter.

Chapter four is a short overview of Flex and YACC. The chapter describes these programming tools and how they are used to build lexical analyzers and parsers.

Chapter five details how the interpreter was implemented. This chapter contains the design specification for the interpreter and a description of the elements involved in building the interpreter. The chapter details each step required to implement the project, and shows how the project begins as a simple desk calculator which evolves into an interpreter for a programming language.
Chapter six shows how the interpreter can be used to study the translation of a high-level language into three-address code. A simple programming example is used to break down the steps involved in using the interpreter to show the three-address code generated during parsing, show the execution of the three-address code, and show the order in which the parser received tokens and the firing actions of the parser as grammar rules were reduced.

Chapter seven contains a short summary of the original project goals, future work that may be done on the project, and the result of the finished project.
2 Description of the Interpreter

This project is an interpreter for a simple C-like programming language. The interpreter can be used as a tool for learning about translation of a high-level language into three-address code. The interpreter can be used in this manner because it is capable of displaying the three-address code generated after parsing a program, showing the execution of the three-address code, and showing the actions of the parser as it reduces grammar productions while parsing a program. The interpreter is designed to execute single-source-file programs and is capable of performing the following actions:

- floating-point computations,
- assignment to variables,
- flow of control statements,
- subroutine calls, and
- recursion.

2.1 Language Definition

The language has the following abilities, and is intended to function much like the C programming language\(^4\). The syntax for the language is therefore a modified subset of the C programming language with the modifications being the exponentiation operator, the reserved word "sub", and print statements.
2.1.1 Operators

The following operators are available.

- Assignment: `=`
- Assignment-addition: `+=`
- Assignment-subtraction: `-=`
- Assignment-division: `/=`
- Assignment-multiplication: `*=`
- Assignment-modulus: `%=`
- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Modulus: `%`
- Exponentiation: `**`
- Less than: `<`
- Greater than: `>`
- Greater than or equal: `>=`
- Less than or equal: `<=`
- Equality: `==`
- Not equal: `!=`
- Increment: `++`
- Decrement: `--`

2.1.2 Flow of Control Statements

The flow of control statements are:

```plaintext
if-else
while
for
```

The syntax for flow of control statements is defined as in the C programming language.

For example, just as in C, else statements are optional and associate with the closest if statement that is not already associated with an else statement. Braces must be used to force any other association.
2.1.3 Subroutines

Subroutines are supported and can return a value if desired. A subroutine returns
a value through a return statement. Subroutine definitions may not be defined inside
other subroutine definitions. A subroutine must be defined before it can be called.

The syntax for a subroutine definition is

```
sub subName(formal parameters)

    statements

}
```

A subroutine is called much like a function is called in C.

```
subName(informal parameters);
```

The number of informal parameters passed in a subroutine call must match the number of
formal parameters declared in the subroutine definition.

2.1.4 Print Statements

A simple print statement is provided. The syntax for a print statement is:

```
print print_list;
```

A print list is either an expression, a quoted string, or a series of expressions and quoted
strings separated by commas. In the following example "called" is a variable.

```
print "The function was called ", called, "times.\n";
```

2.1.5 Return Statements

Return statements are provided. The syntax for a return statement is:

```
return;
return expression;
```
2.1.6 Break Statements

Break statements are provided. The syntax for a break statement is:

    break;

A break statement can only be located within the statements defining a for or a while loop.

2.1.6 Numbers

All numbers are treated as doubles, just as they are in the AWK and PERL programming languages\(^1\)\(^7\). This provides the ability to perform floating point computations. For example, if you make the assignment “avg = 91”, the interpreter will treat “avg” as a double.

2.1.7 Variables

Variables are available for assignment and are also treated as doubles. Because numbers and variables are treated as doubles, there are no declarations of variables, only definitions of variables. For example, to use a variable it is only be necessary to define it as in “i = 0”, or “tmp = 98.6”. A variable's scope may be either global to the program or local to a subroutine.

2.1.8 Comment Convention

Both C and C++ style Comments are provided.

// C++ comment
2.2 Notes

The source programs for this language are only to be contained in a single file. There is no "main()" as in C. The interpreter begins executing where the source code begins.

2.2.1 Scope Rules

The interpreter is a single-pass interpreter. Therefore, the format for writing programs involving global variables should look like the following.

```c
    global variables
    subroutine definitions
    main routine statements
```

For example, in the following program "calls" is a global variable used to count the number of times the subroutine was called during execution.

```c
    calls;       // global "calls" declared and automatically initialized to zero here

    sub recurTest(num){
        calls++;
        if ( num > 0 )
            recurTest( --num );
        print num;
    }

    recurTest( 10 );
    print "The subroutine was called ", calls, "times.\n";
```
If "calls" had been declared below the subroutine definition just before the call to recurTest, the "calls" located inside the subroutine definition would have existed as a completely different local variable to recurTest. The print statement would then have printed a statement telling the user that the subroutine was called zero times.

2.2.2 Invoking the Interpreter

The following is the command line format for invoking the interpreter.

./quad [-Ddvpt] fileName

The characters inside the brackets represent switches and are optional. The following command would be used to execute a program in a file called "programfile".

./quad programfile

The optional switches can be used when invoking the interpreter to indicate to the interpreter whether to perform different tasks. The following switches are defined:

D parse the source code and display the three-address code, no program execution,
d display the three-address code generated after executing the program,
v view the execution of the three-address code,
p show the parser actions in reducing grammar productions,
t show the symbol table,

no switch for normal execution.

Typing "./quad" at the command line will cause a usage error message that lists all the switch definitions. The following invocation executes a program, shows the execution of the three-address code, and displays a dump of the three-address code.
./quad -dv sourceFileName.
The following is a way to display a dump of the three-address code only.

./quad -D sourceFileName.
The following is a way to make the parser show the reductions of grammar productions
during program parsing, and then show the symbol table.

./quad -pt sourceFileName.

2.2.3 System User

Individuals with basic knowledge of programming should be able to use the interpreter very easily. The necessary skills recommended to use the interpreter are familiarity of a Unix based work environment, the ability to use an editor, and the ability to write simple programs in a language such as C, PERL, Pascal, or Java. However, beginning programmers who are familiar with a Unix based system, can use an editor, and are willing to take a few moments to learn the syntax for the programming language will also be able to use the interpreter.

2.3 Examples

The following are several programs that can be executed with the interpreter.

The following program prints a simple message.

\n print "Hello World!\n";

The program gives the following output.

Hello World!
The following program demonstrates the relational operators.

```plaintext
num1 = 4.9;
num2 = 5;

if (num1 == num2)
    print num1, "is equal to ", num2, "\n";
if (num1 != num2)
    print num1, "is not equal to ", num2, "\n";
if (num1 < num2)
    print num1, "is less than ", num2, "\n";
if (num1 > num2)
    print num1, "is greater than ", num2, "\n";
if (num1 <= num2)
    print num1, "is less than or equal to ", num2, "\n";
if (num1 >= num2)
    print num1, "is greater than or equal to ", num2, "\n";
```

The program gives the following output.

4.9 is not equal to 5
4.9 is less than 5
4.9 is less than or equal to 5

The following program demonstrates the increment and decrement operators.

```plaintext
print "Demonstrate post increment\n";
print "a == ", a = 99, "\n";
print "a++ == ", a++, "\n";
print "a == ", a, "\n\n";

print "Demonstrate pre increment\n";
print "a == ", a = 99, "\n";
print "++a == ", ++a, "\n";
print "a == ", a, "\n\n";

print "Demonstrate post decrement\n";
print "a == ", a = 99, "\n";
print "a-- == ", a--, "\n";
print "a == ", a, "\n\n";

print "Demonstrate pre decrement\n";
print "a == ", a = 99, "\n";
print "--a == ", --a, "\n";
print "a == ", a, "\n";
```

The program gives the following output.

Demonstrate post increment
a == 99
a++ == 99
a == 100
Demonstrate pre increment
a = 99
++a = 100
a = 100

Demonstrate post decrement
a = 99
a-- = 99
a = 98

Demonstrate pre decrement
a = 99
--a = 98
a = 98

The following program demonstrate an infinite for loop and the break statement.

```perl
sub testBreak()
{
    for(;;)
    {
        if (a++ == 22)
            break;
        print "a: ", a, "\n";
    }
}
testBreak();
```

The program gives the following output.

```
a: 1
a: 2
a: 3
a: 4
a: 5
a: 6
a: 7
a: 8
a: 9
a: 10
a: 11
a: 12
a: 13
a: 14
a: 15
a: 16
a: 17
a: 18
a: 19
a: 20
a: 21
a: 22
```
The following program demonstrates how the interpreter might be used for floating-point calculations.

```plaintext
principal = 1000.00;
rate = .05;
print "Initial deposit ", principal, "\n", "rate: ", rate, "\n";
print "Year\tAmount on deposit\n";

for (year = 1; year <= 10; year++){
    amount = principal * (1+rate) ** year;
    print year, \t, amount, "\n";
}
```

The program gives the following output.

```
Initial deposit 1000
rate: 0.05
Year Amount on deposit
1 1050
2 1102.5
3 1157.63
4 1215.51
5 1276.28
6 1340.1
7 1407.1
8 1477.46
9 1551.33
10 1628.89
```

The following program calculates the prime numbers between one and four hundred.

```plaintext
isPrim = 1;
for (i = 1; i < 401; i++){
    half = (i/2 +1);
    for ( j = 2; j < half; j++){
        if ((i %j) == 0){ /* if remainder is zero */
            isPrim = 0; /* then number is not prime */
            break;
        }
    }
    if (isPrim)
        print i;
    isPrim = 1;
}
print "\n";
```

The program gives the following output.

```
1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83
89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263
269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359
367 373 379 383 389 397
```
The following program demonstrates a simple recursive subroutine.

```plaintext
sub double(d){
    if(d > 1){
        double(d/2);
    }
    print d;
}

double(1024);
print "\n";
```

The program gives the following output.

```
1 2 4 8 16 32 64 128 256 512 1024
```

The following program is a recursive method to calculate the first ten factorial numbers.

```plaintext
sub fac(num) {
    if (num <= 0)
        return 1;
    return num * fac(num-1);
}

while(i < 10){
    print "factorial of ", i, "is ", fac(i), "\n";
    i++;
}
```

The program gives the following output.

```
factorial of 0 is 1
factorial of 1 is 1
factorial of 2 is 2
factorial of 3 is 6
factorial of 4 is 24
factorial of 5 is 120
factorial of 6 is 720
factorial of 7 is 5040
factorial of 8 is 40320
factorial of 9 is 362880
```

The following is a recursive solution for printing the first twenty Fibonacci numbers.

```plaintext
sub fibo(n){
    if (n <= 1) fibonum = n;
    else fibonum = fibo(n-2) + fibo(n-1);
    return fibonum;
}

for ( i = 0; i < 20; i++){
    a = fibo(i);
    print a;
}
print "\n";
```
The program gives the following output.

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181

The following program is a recursive solution demonstrating Ackermann's function.

    calls;
    sub ack(m, n){
        calls++;
        if (m == 0)
            return (n+1);
        if (m > 0){
            if(n == 0)
                return (ack(m-1, 1));
            else
                if(n > 0)
                    return (ack(m-1, ack(m, n-1)));
        }
    }
    ret = ack(3,3);
    print "calls: ", calls, "\n";
    print "returned: ", ret, "\n";

The program gives the following output.

calls: 2432
returned: 61

The following program demonstrates a recursive solution to the Tower's of Hanoi puzzle.

    sub tower(disks, moves){
        if (disks == 1)
            moves++;
        else{
            moves = tower(--disks), moves);
            moves++;
            moves = tower((disks--), moves);
        }
        return moves;
    }
    print tower(14, 0), "moves\n";

The program gives the following output.

16383 moves
3 Environment

The equipment used during development of this project was a personal computer running the Linux operating system. The project has been completely ported to a Unix workstation in the Computing and Mathematical Laboratory at Texas A&M University - Corpus Christi. The C programming language, Flex and YACC were used to implement the project. The minimum hardware configuration is a Unix based system capable of generating the parser with YACC, generating the lexical analyzer with Flex, and compiling the C source code into the working interpreter. A system capable of building the interpreter will be capable of executing programs written for the interpreter.
4 Overview of Flex and YACC

Flex is a programming tool which is used to generate lexical analyzers. Flex uses a specification containing a set of token descriptions with associated user actions which describe a lexical analyzer. These token descriptions are known as regular expressions and they tell the lexical analyzer which sequence of input characters are to be tokenized.

Flex is a freely available version of Lex which is distributed by the GNU software project. Flex is extremely compatible with Lex and contains few differences from Lex. A few of the differences are that Flex does not support the Lex variable yylineno, and does not allow redefinition of the macros input and output. Additionally, it is claimed that Flex is much more reliable than Lex, and that it generates faster lexical analyzers\textsuperscript{[5]}. The advantage of using Flex on a project is that Flex is available on Linux based systems, and Flex specifications can be easily ported to systems that contain only Lex.

A Flex specification consists of one optional section and three main sections. The main sections are the definition section, the rules section, and the user subroutine section. The layout is shown here\textsuperscript{[5]}:

```%
optional section: C declarations, #include and #define statements
%
definition section: translations and definitions

%%
rules section: regular expressions with associated C user code

%%
user subroutine section: optional user subroutines
```
The input to Flex is the specification file, and the output from Flex is a file called lex.yy.c which contains C source code. The file lex.yy.c is compiled to create the lexical analyzer. The entry point to the lexical analyzer is the function yylex. Figure 4.1 is a graphical representation of how a lexical analyzer is created with Flex.

![Diagram of Flex workflow]

Fig. 4.1. Creation of a lexical analyzer with Flex.
YACC is a powerful parser generator which uses a grammar specification to build a parser. YACC takes some time to learn, but the effort is well worth it. YACC grammars are easily modified and are therefore easier to maintain than the source code required for an equivalent hand written parser. Additionally, once the YACC syntax is understood, a parser can be written in much less time with YACC as compared to the time required to write a parser by hand.

A YACC grammar specification consists of one optional section and three main sections. The main sections are the definitions section, the rules section, and the supporting C-routine section. The layout is shown here:\cite{2}[3][5]:

```
{%
optional section: C declarations, #include and #define statements
%
}
definitions section: token declarations, grammar precedence rules

%%% rules section: the grammar with semantic actions

%%% supporting C-routine section: optional user subroutines
```

The input to YACC is the grammar specification file, and the output from YACC is a file called y.tab.c which contains C source code. The file y.tab.c is compiled to create the parser. The entry point to the parser is the function `yyparse`. Figure 4.2 is a graphical representation of how a parser is created with YACC. Section 5.3.1 describes, in detail, a small grammar specification file.
Fig. 4.2. Creation of a parser with YACC.
5 System Design

The system design was based on stacks and quadruples. The program stack stores quadruples generated during parsing, the control stack is used in calling and return sequences, and the run stack is used for run time calculations during program execution. Figure 5.1 shows a graphical representation of the system design.

![Graphical representation of the system design](image)

**Fig. 5.1.** Graphical representation of the system design.

5.1 Description of Project Elements

There are six main elements involved in the implementation of this project. They are the Flex lexical specification, the YACC grammar specification, the lexical analyzer,
the parser, the symbol table, and the interpreter virtual machine. Figure 5.2 shows an outline of the relationship which exists between the six main elements involved in implementing the project.

![Diagram of interpreter relationships]

Fig. 5.2. Outline of the interpreter relationships.
Flex lexical specification

Function: The function of the lexical specification is to serve as the description which Flex uses to generate the lexical analyzer. The lexical analyzer created is only capable of tokenizing input based upon the token descriptions given in the specification.

Input: The input to the Flex lexical specification is a set of token descriptions known as regular expressions. Regular expressions are essentially a language for specifying patterns that match a sequence of characters[^5].

Output: The output is a file containing the C source code for a lexical analyzer called lex.yy.c. The file lex.yy.c is compiled to create the lexical analyzer called by the C function yy/ex.

Lexical analyzer

Function: The function of the lexical analyzer is to divide the input into meaningful units called tokens. This process is sometimes referred to as tokenizing.

Input: The input to the lexical analyzer is the lines of text from the program source code written in the simple C-like programming language of the interpreter.

Output: The output from the lexical analyzer is the tokens representing the tokenized input.
YACC grammar specification

Function: The function of the grammar specification is to serve as the description which YACC will utilize to generate a parser for the specified programming language.

Input: A set of rules that describe valid statements in the programming language of the interpreter. A rule is also known as a production, and the set of rules are commonly referred to as a grammar. Nearly each rule has a semantic action associated with it. A semantic action is C user code that the parser calls or executes whenever the parser correctly reduces a rule.

Output: The output is a file containing the C source code for a parser which is called y.tab.c. The file y.tab.c is compiled to create the parser which is called by the C function yyparse.

Parser

Function: The function of the parser is to determine if the tokens are logically grouped to form correct statements within any program written in the language for the interpreter. Once a statement has been correctly parsed, the parser generates the quadruples which the interpreter will later execute. The quadruples are generated by the semantic actions associated with each grammar rule.

Input: The input to the parser is the stream of tokens generated by the lexical analyzer.
Output: The output from the parser is the quadruples generated during parsing.

Symbol table

Function: The function of the symbol table is to hold all symbols and any of their attributes. The symbol table functions as a convenient data structure for quickly locating a symbol and looking up, retrieving or storing information about the symbol. The symbol table is implemented in the C programming language as an array of pointers to symbols.

Input: The input to the symbol table is any numbers, variables, strings, or subroutine names encountered in the program source code during lexical analysis and parsing.

Output: The output from the symbol table is any needed symbol attributes such as value, type, or location.

Interpreter virtual machine

Function: The function of the interpreter virtual machine is to execute programs written in the simple C-like programming language of the interpreter.

Input: The inputs to the interpreter virtual machine are the quadruples generated during parsing.

Output: The output from the interpreter virtual machine is either a dump of the three-address code generated during parsing, the display of the execution of the three-address code, or the results from executing the program.
5.2 Procedure Overview

The procedure for implementing this project was similar to a project suggested in the text *Compilers: Principles, Techniques and Tools* by Aho, Sethi and Ullman. The text gives two assignment style outlines for building an interpreter. The second outline describes an interpreter that begins as a desk calculator and evolves into an interpreter for a target language. They describe the desk calculator as essentially an interpreter for expressions, and suggest that constructs be gradually added to the language until an interpreter for the target language is obtained[2]. This project followed a similar method of implementation.

The first step in completing this project was to create a simple desk calculator capable of executing statements involving addition, multiplication, division, and subtraction. This required the creation of the Flex specification for generating a lexical analyzer, the YACC grammar specification for generating a parser, and the necessary C source code required by Flex and YACC for function calls and semantic actions. The next step was to modify the calculator so that it was a small expression interpreter based on quadruples. This required modification of the YACC grammar specification, the creation of the symbol table, and writing the C source code necessary to begin building the interpreter virtual machine. The next and remaining steps in the completion of this project were to gradually refine the existing components in each step until an interpreter was obtained which was capable of executing the language defined above.
5.3 Steps to Completion of the Project

The following steps were followed during the implementation of the project.

1. Implement the desk calculator.
2. Modify the desk calculator so that it was a small interpreter for quadruples.
3. Add variables and variable assignments.
4. Add flow of control statements.
5. Add subroutine definitions.
6. Add the ability to call subroutines and return values.
7. Add the ability to recursively execute a subroutine.
8. Add switches for selection of output.

5.3.1 Implementation of the Desk Calculator

Step one was the creation of the desk calculator. This step essentially involved two components. The first component was the YACC grammar specification which defined rules for the numerical expressions of the desk calculator. The second component was the Flex lexical specification for generating the lexical analyzer.

The first task in the creation of the YACC grammar specification was to design the grammar. A grammar can be thought of as a set of rules that define a programming language, and which is used to determine whether or not a given input is syntactically correct or incorrect\(^5\). The actual grammar for the desk calculator is very simple and is shown in figure 5.3.
Fig. 5.3. Grammar for the desk calculator.

This grammar contains terminals, non-terminals, and productions. The words terminal and token can be considered synonyms when discussing programming languages and grammars. This grammar contains six terminals designated as \texttt{n}, \texttt{NUM}, *, /, +, and -. Terminals within this grammar are represented with upper case letters, or as a single character. There are two non-terminals contained within this grammar, they are \texttt{statement} and \texttt{expr}. The grammar also contains eight simple rules or productions which define a statement and an expression. The grammar indicates that a statement can be defined in one of three ways:

1. A blank,
2. another statement followed by a newline character,
3. another statement followed by an expression followed by a newline character.

The grammar also contains five rules for an expression which is represented by the non-terminal \texttt{expr}. An expression can be a number designated by the token \texttt{NUM}, or any of four mathematical expressions of the form \texttt{expr binary operator expr}, where the binary operator can designate multiplication, division, addition, or subtraction.
Once the grammar was complete, it's syntax was modified so that it could be
understood by YACC, and it was then placed into a YACC grammar specification file.

Figure 5.4 is the full listing of the YACC grammar specification for the desk calculator.

```
1  %{ 
2    #define YYParse double 
3     %} 
4       
5     %token NUM 
6     %left '+' '-' 
7     %left '*' '/' 
8     %right UMINUS /* unary minus */ 
9     %
10    statement: 
11      | statement \n 
12      | statement expr \n   { printf("= %g\n", $2); } 
13   ;
14    
15     expr: 
16      NUM 
17      | expr '*' expr 
18      | expr '/' expr 
19      | expr '+' expr 
20      | expr '-' expr 
21      | '(' expr ')' 
22      | '-' expr %prec UMINUS 
23      { $S = -$2; } 
24    %
```

Fig. 5.4. YACC grammar specification for the desk calculator.

Line two denotes the optional C statements section. Lines four through nine denote the
definitions section, and lines eleven through twenty-three denote the rules section. The
specification does not contain any code in the supporting C-routine section.

YACC utilizes an internal stack to keep track of terminals and non-terminals
during parsing. The stack is normally typed as an integer. The design of the interpreter
calls for the treatment of numbers as doubles. Therefore line two

```
#define YYParse double
```

of figure 5.4 is used to redefine the stack type to a double.
Rules of precedence within a YACC grammar specification flow from lowest levels of precedence to highest levels of precedence\[5\]. In figure 5.4, lines five through eight

```
%token NUM
%left '+' '-'
%left '*' '/'
%right UMINUS /* unary minus */
```

denote token definitions for the grammar, and lines six through eight additionally define rules of precedence. For example, line six

```
%left '+' '-'
```

defines the tokens \"+\" and \"-\", indicates that they are left associative, and defines them to be at the lowest level of precedence. Additionally, line eight

```
%right UMINUS /* unary minus */
```

defines the pseudo token UMINUS, representing unary minus, to be right associative and at the highest level of precedence.

The production definitions for a statement and an expression are still the same as in the original grammar of figure 5.3, but YACC understands a more concise grammar syntax. Each set of related productions begins with a non-terminal, and ends with a semicolon. The bar separating each related production can be thought of as \"or\". In figure 5.4, lines eleven through fourteen

```
statement: |
    statement '\n'
    |    statement expr '\n' { printf("= %g\n", $2); }
|

```

denote the grammar productions for a statement, and lines sixteen through twenty-three
denote the grammar productions for an `expr`. The two final rules in the set
were added so that an expression may be isolated within parenthesis or preceded by a
subtraction operator.

On the right side of most of the productions of figure 5.4, there are some
embedded program fragments located within braces.

```c
{ printf("= %g\n", $2); }
{ $2 = $1; }
{ $2 = $1 * $3; }
{ $2 = $1 / $3; }
{ $2 = $1 + $3; }
{ $2 = $1 - $3; }
{ $2 = $2; }
{ $2 = -$2; }
```

The code within the braces is referred to as a semantic action. A semantic action is user
code that the parser will execute when a particular rule is reduced correctly\(^2\). A
semantic action can refer to a non-terminal on the left hand side of a production by
referencing it as $S$. Similarly, a semantic action can refer to any terminal or non-
terminal on the right hand side of a production by referencing the symbol as $S_i$, where $i$
represents a symbol location beginning at one. For example, the `$2` within the semantic
action located on line thirteen

```c
statement expr 'n' { printf("= %g\n", $2); }
```

refers to `expr`, because it is the second non-terminal on the right hand side of that
particular production. The semantic action on line thirteen is a simple C language `

```
printf```

33
statement which tells the parser to print an equal sign followed by the value of the expression.

The second task in the completion of step one was to create the Flex lexical specification. Figure 5.5 is the full listing of the lexical specification for the desk calculator. Flex used this specification to build a lexical analyzer which specifically recognizes numbers, white space, and newline characters.

```
1  %{
2    #include "y.tab.h"
3    extern double yyval;
4    int yylineno = 1;
5  %}
6
7  \%
8  [ \t]              { ; \_\_\_ ignore whitespace */ }  
9
10 \n
11              { yylineno++;  
12                  return yytext[0]; }  
13
14 [0-9]+  
15 (([0-9]*\.\[0-9]+)  
16      ([0-9]+\.\[0-9]*))  
17              { yyval = atof(yytext);  
18                  return NUM;  
19              }  
20
21              { return yytext[0]; }  
22\%
```

Fig. 5.5. Flex lexical analyzer specification.

Lines two through four denote the optional section. Line six denotes the definition section, and lines eight through twenty-one denote the rules section. The specification does not contain any code in the user subroutine section. Line two tells Flex to include the file y.tab.h. The file y.tab.h was auto-generated by YACC and contains token definitions which are defined numerically with C #define statements. There are only two
declared tokens in the declarations section of the YACC grammar specification that are not single characters, so the file y.tab.h only contained two statements.

```c
#define NUM 257
#define UMINUS 258

Line three of figure 5.5
extern double yylval;
tells Flex that the YACC variable yylval is a double. Any token that is returned by Flex has its value stored in the variable yylval. Lines four, ten, and eleven
```n```c
int yylineno = 1;
\n\n{ yylineno++;
  return yytext[0]; }
```n
take care of the fact that Flex does not keep track of line numbers. Line ten counts line numbers and line eleven tells Flex to return a newline character as a token. Line eight
```c
[ \t]         { ; /* ignore whitespace */ }```
contains a regular expression defining white space and tabs. The action associated with this regular expression is to do nothing. The do nothing action is indicated by the semicolon. Lines thirteen, fourteen, and fifteen
```c
[0-9]+       |  \n([0-9]*\.[0-9]+)   |  (\[0-9]+\.[0-9]*)   |
```
denote regular expressions for recognizing numbers. Line thirteen denotes a regular expression which defines whole numbers or integers. Line fourteen and fifteen denote regular expressions which define real or floating point numbers. The action on line fifteen
```c
yylval = atof(yytext);
```n
tells the lexical analyzer to call the C function `atof` to convert the text value of a tokenized number into its numeric value and store the value in the variable `yyval`. The period and action on line twenty

```
  { return yytext[0]; }
```

tells Flex to return all other single input characters as tokens.

The final requirements necessary to implement the desk calculator were a C function call to invoke the parser, the definition of the function `yyerror`, and the compilation of the source files. Figure 5.6 shows the contents of the file `main.c` which contains the function call to `yyparse`, and the definition of the function `yyerror`. The function `yyparse` is the entry point to the generated parser. YACC requires that the function `yyerror` be defined because `yyerror` is called whenever YACC encounters an error during parsing.

```c
int main(void)
{
  yyparse();
  return 0;
}

yyerror(char *s)
{
  extern int yylineno;
  extern char *yytext;
  printf("%d: %s at %s\n", yylineno, s, yytext);
}
```

Fig. 5.6. Contents of the file `main.c` during step one.

In order to execute the desk calculator, the files `main.c` `lex.yy.c` and `y.tab.c` were compiled to form the executable file called `a.out`. The desk calculator was invoked by typing `./a.out`. Figure 5.7 shows some output from the desk calculator.
$ ./a.out
  .2+3.5
  = 3.7
  -2/.2
  = -10
  2+7*8
  = 58
  (2+7)*8
  = 72
  2+7*8+3/3
  = 59

Fig. 5.7. Output from desk calculator during step one.

5.3.2 Modification of the Desk Calculator into a Small Interpreter for Quadruples

Step two began the transformation of the desk calculator into an interpreter based on quadruples. A quadruple is a form of three-address code which is defined as a record structure with four fields. The fields are normally pointers to symbols in the symbol table. There is a field for a numeric operator code representing the operation to be performed, a field representing the first argument to the operator, a field representing the second argument to the operator, and a field representing the result of the operation.

Figure 5.8 is the C structure which defines quadruples for the interpreter.

```c
typedef struct quad { /* a quadruple or three address code */
  int op; /* opcode */
  Symbol *arg1;
  Symbol *arg2; /* not used by unary statements or param */
  Symbol *result; /* not used by param */
} Quad;
```

Fig. 5.8. Definition of a quadruple.

First, the symbol table was created because the fields of a quadruple are pointers into the symbol table. A symbol is also a record structure with members representing the attributes of a token or terminal symbol. Figure 5.9 shows the C structure which defines symbols for the interpreter.
typedef struct Symbol {
    char     *name;    /* store yytext */
    int      tokentype;  /* NUM, VAR, SUB */
    int      lineno;     /* yylineno */
    double   val;       /* symbol value */
    struct Symbol *next;
} Symbol;

Fig. 5.9. Definition of a symbol.

The symbol table is defined as an array of pointers to symbols. This data structure is
sometimes referred to as a hash table. The following two C statements define the symbol
table.

#define TABLESIZE 32
Symbol *SymTable[TABLESIZE];  /* the symbol table */

There are two primary functions used to manage the symbol table. The first is a
lookup function called SymbolExists, which is used to determine if a symbol exists within
the symbol table. The second is a function called InsertSymbol which is used for
inserting new symbols into the symbol table. There are three additional symbol table
management functions. The first is called salloc, which is used for allocating symbol
space. The second is called StrSpace, which is used for allocating string space. The third
is called HashIt, which is used for locating the index into the array of pointers to symbols.
Figure 5.10 shows a listing of all of the symbol table management functions.
Symbol * salloc(void)
/* function to allocate symbol space to point to */
{
    return (Symbol *) malloc(sizeof(Symbol));
};

int HashIt(char *s)
{
    int val;
    unsigned hashval;
    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + TABLESIZE * hashval;
    val = hashval % TABLESIZE;
    return val;
};

Symbol * SymbolExists(char *str)
/* function looks up a symbol and returns a pointer to it if found */
{
    Symbol *ptr;
    ptr = SymTable[HashIt(str)];
    while (ptr != NULL)
        if (strcmp(ptr->name, str) == 0)
            return ptr;
        else
            ptr = ptr->next;
    return 0;
};

Symbol * InsertSymbol(char *str, int type, double d)
/* inserts a symbol into the symbol table */
{
    int index;       /* index into the symbol table */
    Symbol * nsp;   /* new symbol pointer */
    index = HashIt(str);  /* find the array index */
    nsp = salloc();     /* make space for nsp to point at */
    nsp->name = StrSpace(str);
    nsp->tokentype = type;
    nsp->lineno = yylineno;
    nsp->val = d;
    nsp->next = SymTable[index]; /* point to what SymTable points to */
    SymTable[index] = nsp;     /* point SymTable to new symbol */
    return nsp;
};

char * StrSpace(char *s)
/* function makes space for char pointer to point at */
{
    char *p;
    p = (char *) malloc(strlen(s)+1);   /* +1 for '\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
};

Fig. 5.10. Symbol table management functions.
Before the YACC grammar specification was refined, some of the interpreter virtual machine was implemented. The first component needed by the virtual machine was a stack to store the quadruples generated during parsing. After parsing, the stack functions as the program and the quadruples function as the object code of the interpreter. The following C statement defines a program stack which contains 512 quadruples.

Quad program[512];

The next component needed by the virtual machine was a table of virtual machine operator codes. Figure 5.11 is a listing of the file opcodes.h.

/* opcodes.h */
/* table of virtual machine opcodes */
#define _mul           1
#define _div           2
#define _add           3
#define _sub           4
#define _neg           5

Fig. 5.11. Virtual machine operator codes.

Additionally, the virtual machine required the definition of a function for loading quadruples onto the program stack. Figure 5.12 shows the definition of the Load function.

void Load(int op, Symbol *arg1, Symbol *arg2, Symbol *result)
/* function to load a quadruple */
{
    program[lpo].op = op;
    program[lpo].arg1 = arg1;
    program[lpo].arg2 = arg2;
    program[lpo++].result = result;
}

Fig. 5.12. Definition of Load function.
The \textit{Load} function is located in the file \texttt{interp.c}. The \textit{Load} function receives four arguments when it is called. The first is the numeric operator code representing the operation to be performed with the quadruple. The second and third arguments are pointers to the symbols which are the arguments to the operator. The final argument is a pointer to the symbol in which the result of the operation should be stored. The following statement defining the load program counter is also included in the file \texttt{interp.c}.

\begin{verbatim}
static int lpc = 0;
\end{verbatim}

The variable \texttt{lpc} is shown in the \textit{Load} function of figure 5.12. The load program counter is used to keep track of the program counter during parsing.

The next task was to refine the YACC grammar specification. The YACC stack type was changed to accommodate symbols. Also, the semantic actions embedded within the grammar were rewritten so that quadruples were pushed onto the program stack during parsing. Figure 5.13 shows the refined grammar specification.

\begin{verbatim}
1 \%
2 \#include <stdio.h>
3 \#include "type.h"
4 \#include "interp.h"
5 \#include "symtable.h"
6 \#include "opcodes.h"
7 \%
8 \%union { \ /* redefine the yacc stack type */
9   double val; \ /* for numeric values */
10   Symbol *sp; \ /* symbol pointer */
11 \}
12
13 \%token <val> NUM
14 \%type <sp> expr
15 \%left '+' '-'
16 \%left '*' '/'
17 \%nonassoc UMINUS \ /* unary minus */
18
19 \%
\end{verbatim}

Fig. 5.13. YACC grammar specification modified for quadruples (part 1 of 2).
Lines eight through eleven

```c
union {
  /* redefine the yacc stack type */
  double val; /* for numeric values */
  Symbol *sp; /* symbol pointer */
}
```

denote the method for changing the YACC stack type. The union definition tells YACC that its internal stack type is now redefined to be a C union containing data objects which are typed as either symbol pointers or doubles. The characters `<val>` on line thirteen

```c
	%token <val> NUM
```
tell YACC that the token `NUM` is to be treated as a data object of type double. The characters `<sp>` on line fourteen

```c
	%type <sp> expr
```
tell YACC that the non-terminal `expr` is to be treated as a symbol pointer.
Because quadruples point into the symbol table, temporary symbols must be created and entered into the symbol table as they are parsed\(^2\). The following semantic action on line twenty-six of figure 5.13 creates a new temporary symbol by calling the function \texttt{InsertSymbol}.

\[
\texttt{\$\$ = InsertSymbol("", \texttt{NUM}, 0.0);}
\]

\texttt{InsertSymbol} inserts a new symbol into the symbol table, sets its name to a blank, sets its token type as \texttt{NUM}, and sets its initial value to zero (0.0). The assignment to \texttt{\$\$} stores the symbol location into the non-terminal \texttt{expr}, on the left hand side of the rule because the function \texttt{InsertSymbol} returns a pointer to the newly inserted symbol. The following semantic action on line twenty-seven of figure 5.13 is used to load a quadruple onto the program stack during parsing.

\[
\texttt{Load(_mul, \$1, \$3, \$\$);}
\]

The code for the \texttt{Load} function is shown in figure 5.12. The \texttt{Load} function takes a numeric operator code and the values of the terminals or non-terminals of the associated production as arguments. The values of the terminals or non-terminals are pointers into the symbol table. These same semantic actions to create and insert temporary symbols, and to load quadruples onto the program stack, were inserted into the grammar productions defining multiplication, division, addition, subtraction, and unary minus. These semantic actions are executed each time one of the productions is reduced correctly.

The next task was to make two small changes to the Flex specification for the lexical analyzer. The first change was made so that Flex knows that the YACC stack type