T. C.
HALIC UNIVERSITY
THE INSTITUTE OF SCIENCE
COMPUTER ENGINEERING

DESIGN AND IMPLEMENTATION OF INTERPRETERS

MASTER THESIS

Oğuz KARAN

Supervisor
Prof. Dr. Ali OKATAN

ISTANBUL 2005

T. C.
HALIC UNIVERSITY
THE INSTITUTE OF SCIENCE
COMPUTER ENGINEERING

DESIGN AND IMPLEMENTATION OF INTERPRETERS

MASTER THESIS

Oğuz KARAN

Supervisor
Prof. Dr. Ali OKATAN

ISTANBUL 2005

T.C.

HALİÇ ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ MÜDÜRLÜĞÜNE

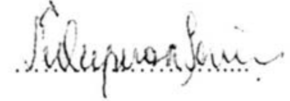
Oğuz KARAN tarafından hazırlanan " **Design and Implementation of Interpreters** " adlı bu çalışma jürimizce Yüksek Lisans Tezi olarak Kabul Edilmiştir.

Kabul (Sınav) Tarihi : 03.01.2006

(Jüri Üyesinin Ünvanı , Adı , Soyadı ve Kurumu) : İmzası :

Jüri Üyesi : Prof.Dr. Ali OKATAN

Jüri Üyesi : Prof.Dr. Süleyman SEVİNÇ



Jüri Üyesi : Yrd.Doç.Dr. Rıfat ÇÖLKESEN
(Beykent Üni.Öğr.Üy.)

To My Parents

TABLE OF CONTENTS

TABLE OF CONTENTS	i
PREFACE	iii
ACKNOWLEDGEMENTS	iv
ÖZET	v
ABSTRACT	vi
LIST OF FIGURES	vii
1. INTRODUCTION	1
1.1 Translators and Compilers	1
1.2 Interpreters	1
1.3 Comparison of Compilers and Interpreters	2
1.4 Phases of compilation process	3
1.5 Importance of interpreter / compiler design	4
1.5.1 Wide applicability of interpreter / compiler construction	5
1.5.2 Useful algorithms while constructing Interpreter / Compiler	5
1.6 History of Interpreter / Compiler Design And Implementation	5
1.6.1 1945-1960:	5
1.6.2 1960-1975:	6
1.6.3 1975-Present:	6
1.7 Notations of Syntax Definition	7
1.8 BNF Notation	7
1.9 Abstract Syntax Tree and Annotated Abstract Syntax Tree	10
1.10 Symbol Table	11
2. A SIMPLE MANUAL CALCULATOR (SIMPLE DEMO INTERPRETER/COMPILER)	13
2.1 The Grammar for the simple manual calculator	13
2.2 Main module for the simple manual calculator	14
2.3 Scanning or Lexical Analysis for the simple manual calculator	14
2.4 Parsing or Syntax Analysis for the simple manual calculator	15
2.5 Interpretation and code generation for the simple manual calculator	17
3. LEXICAL ANALYSIS OR SCANNING	18
3.1 Regular Expressions	18
3.1.1 Using the Regular Expressions	19
3.1.2 Theory of Regular Expressions	23
3.1.3 Regular Expressions in C and C++	25
3.2 Lex / Flex programs	26
3.2.1 Rules Section	27
3.2.2 Codes Section	28
3.2.3 Important Global Variables in the lex/flex Source Code	29
3.2.4 Compiling and Executing the lex/flex source file	29
3.2.5 Lexical Analysis Process with yylex Function	30
3.2.6 Using lex/flex in Projects	30
3.3 Manual Lexical Analysis	31
3.4 Object Oriented Manual Lexical Analysis by using C++	31
4. SYNTAX ANALYSIS OR PARSING	34
4.1 yacc/bison Programs	34
4.1.1 Organization of yacc/bison source (input) file	34
4.1.2 Process of yacc/bison	36

4.1.3 Expression and Atom Types in yacc/bison 37
4.1.4 Using C++ codes with lex/flex and yacc/bison 38
4.2 Representing the Syntax Tree by Using C 39
4.3 Constructing a Syntax Tree by using yacc/bison 40
4.4 Error Handling in yacc/bison 42
4.5 Manual Syntax Analysis 42
4.6 Object Oriented Manual Syntax Analysis by using C++ 43
5. SAMPLE INTERPRETER 44
5.1 Design of Sample Interpreter 44
5.2 Implementation of Sample Interpreter 49
5.3 Properties of Sample Interpreter 49
REFERENCES 51
6.1 APPENDIX A 53
6.1.1. main.c file 53
6.1.2. scanner.h file 53
6.1.3. scanner.c file 54
6.1.4. parser.h file 56
6.1.5. parser.c file 56
6.1.6. backend.h file 58
6.1.7. backend.c file 58
6.2 APPENDIX B 60
6.2.1. Simple use of POSIX regex functions 60
6.2.2. scanner.h file 61
6.2.3. scanner.cpp file 64
6.3 APPENDIX C 68
6.3.1. parser.h file 68
6.3.2. parser.cpp file 68
6.4 APPENDIX D 71
6.4.1. interpreter.l file 71
6.4.2. interpreter.y file 73
6.4.3. interpreter.h file 81
6.4.4. interpreter.cpp file 83
6.4.5. interpreter.hpp file 84

PREFACE

Interpreter programs are system programming softwares and also design and implementation needs advanced knowledge and experience. These developments were driven by the advent of new programming paradigms. Learning this paradigm is important for solving other kinds of problems. For example; programs that convert any file format into another can be written by using the compiler/interpreter design paradigm. Furthermore, with this paradigm, important and advanced data structures must be used. For that reason, programmers have a good experience in data structures.

In this thesis, it is aimed learning design and implementation of compilers/interpreters. Furthermore, sample interpreter uses this paradigm in details. This interpreter is the detailed generic (skeletan) interpreter and also shows the way to design another compilers/interpreters for any programmer.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Prof.Dr.Ali OKATAN for his supervision and encouragement during my thesis study. He has been so kind and patient in all my desperate times. His trust, support, contributions and understanding motivated me and let this thesis come to end.

I am grateful to my parents for their support and encouragement throughout my graduate study.

Last but not least, I would like to thank to Barış Ağca, Ersin Turgut, Kadir Kurt, Kayra Önder, Sercan Uzun, Yavuz Karan, Prof.Dr. Yavuz Gündüzalp and Öğr.Gör.Dr.Bülent Ayanlar for all their friendship, encouragement, support and all the nice times we spent together.

ÖZET

YORUMLAYICI TASARIMI VE KODLANMASI

Karan, Oğuz

Tez Yoneticisi: Prof. Dr. Ali Okatan

İstanbul 2005, 85 sayfa

Bu tezde yorumlayıcı tasarımı detaylı olarak ele alınmış ve her aşaması ayrı ayrı anlatılmıştır. Yorumlayıcı, programın kaynak kodunu anlamlandırıp çalıştıran programdır. Bu yazılım, makineden bağımsız olarak kod yazmaya ve geliştirilmesine olanak sağlar. Bu tarz yazılan programlar için işletim sistemi düzeyinde çalışan aslında yorumlayıcı programın kendisidir. Bu duruma örnek; Java programlama dili ve .NET platformunda geliştirilen programlar verilebilir.

Yorumlayıcı sistem programlama yazılımlarından biri olup, tasarımı ve geliştirilmesi ileri düzeyde bilgi birikimi ve tecrübe gerektirmektedir. Karışık bir yapısı olan yorumlayıcıyı tasarlamak ve geliştirmek için teorik bilginin yanı sıra pratik bilgi de gerekmektedir.

Bu çalışmada ayrıca detaylı (genel) iskelet yorumlayıcı oluşturulmuş ve son bölümde tasarımı ve kodları anlatılmıştır.

Anahtar Kelimeler: Derleyici, Yorumlayıcı

ABSTRACT

DESIGN AND IMPLEMENTATION OF INTERPRETERS

Karan, Oğuz

Supervisor: Prof. Dr. Ali Okatan

Istanbul 2005, 85 pages

In this thesis, interpreter design is discussed in details and all phases are explained one by one. Interpreter is a program that meaning and executes the source code. For this kind of programs, real process in operating system's level is the interpreter. Programs that are implemented in Java programming language and in .NET platform can be given as example.

Interpreter programs are system programming softwares and also design and implemantation needs advanced knowledge and exprience. There should be applied knowledge as well as theoretical information for designing and implementing of an interpreter.

In this study, additionally detailed generic(skeletal) interpreter is developed and in last chapter design and implementation of this interpreter is explained.

Key words: Compiler, Interpreter

LIST OF FIGURES

1. Figure 1.1……………………………………………………………………………..4
2. Figure 1.2……………………………………………………………………………11
3. Figure 2.1……………………………………………………………………………13
4. Figure 2.2……………………………………………………………………………17
5. Figure 3.1……………………………………………………………………………25
6. Figure 3.2……………………………………………………………………………27
7. Figure 3.3……………………………………………………………………………33
8. Figure 4.1……………………………………………………………………………49
9. Figure 4.2……………………………………………………………………………51
10. Figure 4.3…………………………………………………………………………...53
11. Figure5.1……………………………………………………………………......63, 64
12. Figure5.2……………………………………………………………………………65

1. INTRODUCTION

1.1 Translators and Compilers

Software that transforms one program code that is written in any programming language into another programming language is called *translator program* or *translator*. The language that is being transformed is called *source language* and the resulting one is called *target language* [1].

For example, a program that transforms *Pascal* code to *C* code is a translator. Particularly, if the source is the high level or mid level language and the target is the low level language, (*Assembly language* or *pure machine language*), then the translator is called *compiler*, however the program that translates the assembly language to pure machine language is called *assembler*. Although assemblers transform source to target language, they are not technically translators. *Linkers* that link the object codes are similar to translators. Also, the loader of the operating system is similar to translators.

From the pragmatic point of view, the translator defines the semantics of the programming language; it transforms operations specified by the syntax into operations of the computational model [18].

1.2 Interpreters

Interpreter is a program that executes the source code partially. For example, *Interpreter* takes a line from the source text and establishes the action and processes it. In other words, the process that executes at operating system's level is actually interpreter program. Interpreter realizes the meaning of source code and processes the code. Some traditional languages work with interpreters. For example; *Basic*,

Pascal, APL. Several *script languages* like *Matlab*, *AWK* are also processed with interpreters.

1.3 Comparison of Compilers and Interpreters

Compilers and *Interpreters* are complex programs. It is generally believed that only most advanced system programmers are privacy to this arcane art . That is the part of the mystique of being considered the Grand Guru of the programming department [2].

Interpreters are normally written in high-level language and will therefore run on most machines, whereas compilers generate object code for specific machine architecture. In other words portability is increased for interpreters. As *interpreters* interpret the code line by line, they cannot completely control the error. Generally, when they first encounter an error, they terminate the program.

Since *interpreters* do not generate target code, they are not considered as *translators*. There is no difference between using *compiler* and *interpreter*. In both cases the program text is processed into an intermediate form which is then interpreted by some interpreting mechanism. As the *interpreters* are programs that interpret the source code, the execution of the code is relatively slower than compiler [1].

1.4 Phases of compilation process

In fact, compiler makes analysis and syntheses. Compiler first analyzes the source code, then makes processes for generating target code. The following paragraphs define the general phases of the compilation process:

1. *Lexical analysis* or *scanning:* This phase analyses the character string presented to it and divides up into tokens that are legal members of the vocabulary of the language in which the program is written. In this phase, compiler can generate error messages if the character string is not pursuable into a string of legal tokens [3].
2. *Syntactic Analysis or parsing*: This phase processes the sequence of tokens and produces an intermediate-level representation, such as abstract parse tree or abstract syntax tree and a symbol table that records the identifiers used in the program and their attributes. Parsing can occur in two basic fashions: top-down and bottom-up [17]. In this phase compiler can generate error messages if the token string contains syntax errors or the misuse of the operator [3]. For parse trees, there is an additional phase called context handling that determines and place annotations or attributes for any node in tree. This version of the tree is generally called *Annotated Abstract Syntax Tree*. Abstract Syntax Tree (AST) is often called for Annotated Abstract Syntax Tree [1].
3. *Semantic Analysis;* this phase processes the tasks using the symbol table and AST. In this phase; compiler can produce error messages such as type compatibility problem in the language.

4. *Intermediate Code Generation:* The code generation begins at this stage. This code is used for optimization. Optimizing phase is called *code optimization.*
5. *Code Generation:* This phase produces the target code from intermediate or optimized intermediate code.

The 1st, 2nd, and 3rd phases are called the *front-end* of the compilation process others are called *back-end* of the compilation process. There are additional tasks in the middle of these phases. These are the general ones. Phases of compilation process are shown in Figure 1.1.

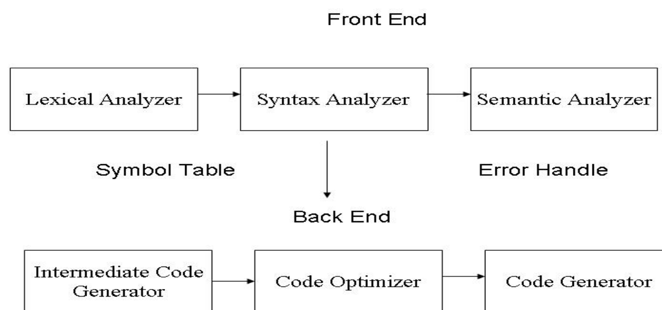


Figure 1.1 The general phases of compilation process [4].

1.5 Importance of interpreter / compiler design

Studying compiler construction is a good idea because:

- Given its close relation to file conversion, it has wider application than just compilers.
- It contains many generally useful algorithms in a realistic setting.

1.5.1 Wide applicability of interpreter / compiler construction

Interpreter / compiler construction paradigm can be applied to many other problems. Many problem can easily be solved effectively by using this technique. For example, when creating new file formats, file conversion problems, etc. If the data can be obtained by writing a grammar for it, it can be useful to obtain this data by using this technique as a parser generator. Then *parser* can be generated automatically. Such techniques can for example, be applied to rapidly create 'read' routines for HTML files, Postscript files, etc. Examples of file conversion systems that have profited considerably from compiler construction techniques are TeX text formatters which convert TeX text to dvi format and Postscript interpreters which convert PostScript text to instructions for a specific printer [1].

1.5.2 Useful algorithms while constructing Interpreter / Compiler

Interpreter / compiler construction techniques contain useful data structures and algorithms. Examples include hashing, garbage collections, graph algorithms, and trees etc. Using of these algorithms is educationally more valuable than their isolated study [1].

1.6 History of Interpreter / Compiler Design And Implementation

Three periods can be distinguished in the history of interpreter / compiler design and implementation [1]:1945 – 1960, 1960-1975 and 1975-present.

1.6.1 1945-1960:

In the 1950s, the earliest programmers prided themselves on doing their work without any assistance, and their work was tedious—in the extreme [16].

During this period languages have been developed relatively slow and machines were idiosyncratic. The primary problem was how to generate code for a given machine. Proponents of high-level languages feared, not without reason that idea of high-level programming would never catch on if compilers produced code that was less efficient than what assembly programmers produced by hand. The first Fortran compiler (Sheridan, 1959) optimized heavily and was far ahead of its time in that respect [1].

1.6.2 1960-1975:

During this period language designers believed that having a compiler for a new language was more important than having one that generated very efficient code. At the same time, studies in formal languages revealed a number of powerful techniques that could be applied profitably in *front-end* construction, notably in parser generation [1].

1.6.3 1975-Present:

From 1975 to the present, both the number of new languages proposed and the number of different machine types in regular use decreased, which reduced the need for quick –and – simple compilers for new languages and / or machines. The greatest turmoil in language and machine design being over, people began to demand professional compilers that were reliable, efficient, both in use and in generated code, and preferably with pleasant user interfaces. This called for more attention to the quality of the generated code, which was easier now, since with the slower change in machines the expected lifetime of a code generator increased [1].

1.7 Notations of Syntax Definition

There are number of grammar forms recommended for describing the syntax of programming languages. Most popular one is the BNF (Buckes Naur Form) notation. BNF notation first was designed in 1958 by John Buckes. BNF notation was used for describing the syntax of Algol58. Peter Naur improved this notation. For this reason, this notation is called BNF. BNF and its derivations are used for describing the most programming languages. Improved version of the BNF is called EBNF (Extended BNF). EBNF was standardized by ISO with the reference number ISO / IEC 14977:1996(E).

1.8 BNF Notation

BNF notation contains three components:

1. Production
2. Non-terminal symbols
3. Terminal symbols (atoms)

Any line of the syntax rule is called *production.* In BNF notation all non-terminal symbols replaced as long as the atom encounters. For example

S ::= S digit | digit

Left - hand side of the symbol (S) ::= is the symbol that is described. Right – hand side is the syntactic construct. All of the line is called production. In some BNF derivations, symbol ":" is called for instead of symbol "::=".

In BNF notation symbol "|" (pipe) means *or.* For example, for

S ::= S digit | digit

S can be a digit *or* S digit. The BNF production can be recursive. In some BNF derivation, instead of “|” for *or* the part of *or* is written to the below. For example

```
S ::= digit
      S digit
```

This kind of notation is used in C/C++ standards.

A non-terminal symbol can be described by another or directly by using a terminal symbol. For

S ::= S digit | digit,

digit is also a non-terminal symbol. All non-terminal symbols must be described. For production, the syntax must be of the following form:

```
S :
        S digit
        digit

digit:
        0
        1
        3
        4
        5
        6
        7
        8
        9
```

For this rule, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 are terminal symbols (atoms).

For any rule, terminal symbols must be distinguished from non-terminal symbols. In original BNF notation terminal symbols are directly written and non-terminal symbols are written in the middle of the angular parentheses. For example:

S ::= <S> <digit> | <digit>

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Some BNF derivations (C/C++/C# standards) any symbol are written directly but atoms are written in bold or italic.

Optional elements are written in the middle of []. In C/C++ standards optional elements are written including *opt* sub index. Original BNF contains no optional element concept.

For example, typical Pascal program can be written in BNF notation as follows.

```
<program> ::= program
              <decleration-sequence>
              begin
              <statement-sequence>
              end.
```

1.9 Abstract Syntax Tree and Annotated Abstract Syntax Tree

The *syntax tree* of a program text is a data structure which shows precisely how the various segments of the program text should be viewed in terms of the grammar [1].

The output of the syntax analyser and semantic analyser phases is sometimes expressed in the form of a decorated AST. This is a very useful representation, as it can be used in clever ways to optimize code generation at a later stage. [21].

Since this tree is used for parsing, it is also called *parse tree.* Parsing is also called *syntax analysis.* Nodes of the parse tree are the non-terminal symbols and the leaves of these are the terminal symbols. For example a for grammar rule below;

```
expression:
        additive_expression
        | factor_expression
additive_expression:
        additive_expressipn '+' factor_expression
        | factor_expression

factor_expression:
        T '*' T
        | T
```

Derivation of the syntax tree,

T * T + T * T

is shown in Figure 1.2:

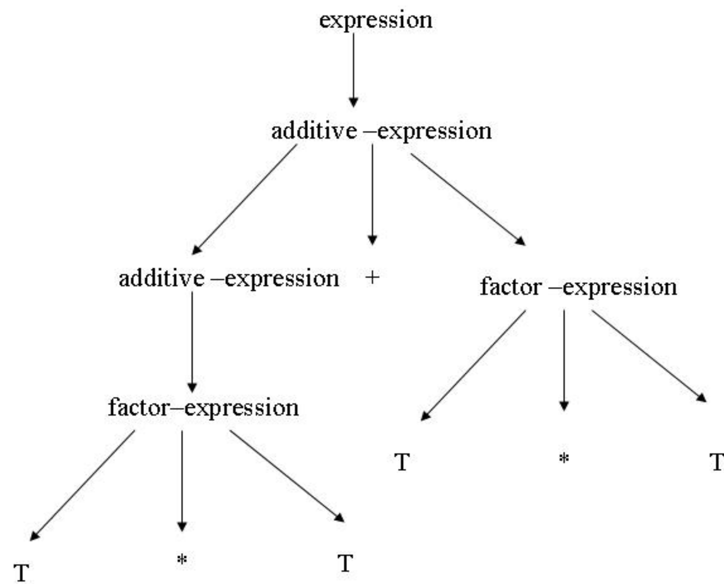


Figure 1.2 Syntax tree of the grammar rule as given above

The exact form of the parse tree as required by the grammar is rarely the most convenient one for further processing, so usually a modified form of it is used, called an *abstract syntax tree* or *AST.* Detailed information about the semantics can be attached to the nodes in this tree through *annotations*, which are stored in additional data fields in the nodes; therefore the term *annotated abstract syntax tree.* Since un annotated ASTs are of limited use, ATSs are always more or less annotated in practice, and the abbreviation *AST* is used also for annotated ASTs [1].

1.10 Symbol Table

Compilers and Interpreters build and maintain a data structure used throughout the translation process. This structure is commonly called the *symbol table* and it is where information about many of the source program's token is kept [2].

Symbol table is used for recording the name, current value and other attributes of any variable. *Symbol table* must also able to design for adding or getting variable data when a variable is encountered.

A C compiler, for example, stores the variable and structure names, labels, enumeration tags, and all other names used in the in its symbol table [19].

There are three operations on *symbol table*:

- Add (Insert)
- Delete
- Search

Generally *search operation* is processed frequently than the *add operation.* For this reason, data structure must be effective for *search operation*. *Hash Table* is a good solution for this problem. *Hashing with chain* method can be used.

Chains provide a good solution to the overflow problem that arises when hashing is used. Rather than placing an element into a bucket other than its home bucket, it is maintained as chains of elements that have the same home buckets [20].

First, variable name is passed to *hash function* the result is the *hash index,* then is searched in the related *linked list.* Also *binary trees* can also be used for symbol tables.

2. A SIMPLE MANUAL CALCULATOR (SIMPLE DEMO INTERPRETER/COMPILER)

Implementation of the simple manual calculator project consists of four modules. These are *scanner.c, parser.c, backend.c* and *main.c*. All modules except *main.c* have their own headers, *scanner.h, parser.h, backend.h*.

2.1 The Grammar for the simple manual calculator

For a simple manual calculator, the expressions are based on a *fully parenthesized expression with operands of one digit.* This makes *parsing/syntax analysis* simple rand avoids details. The grammar of this calculator is shown in Figure 2.1.

```
expression:
        digit
        | '(' expression operator expression ')'

operator:
        '+'
        | '*'

digit:
        0 | 1 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Figure 2.1 Grammar for simple manual calculator

To simplify things even further, this calculator has only two operators [1]. Sample derivations that the grammar produces in Figure 1.3 are

```
3
(1 + 2)
(1 * ((3 + 5) * 7))
etc.
```

Also this allows white space, including tabs and new lines, in the input.

2.2 Main module for the simple manual calculator

The main module of the program file is *main.c* as shown in Appendix A.1. In this project, source type can be either a string from console or a file. For the file is shown in Appendix A.1, the source type is file (*sample.dat*). For this reason the sourceType field of the g_source global variable, which is defined in scanner.c and has a type *SOURCE* type that is declared in scanner.h, must be assigned to the appropriate type either string form console or a file in main module.

After the initializations of the variable *g_source, Parse* function which is defined in *parser.c* and declared in *parser.h*, is called with a node variable. Afterwards *Process* function called for interpreting code or generating an executable code.

2.3 Scanning or Lexical Analysis for the simple manual calculator

The scanning module of the simple manual calculator program is *scanner.c* and its own header is *scanner.h*. The contents of the *scanner.h* file are shown in Appendix A.2, and the contents of the *scanner.c* file are shown in Appendix A.3. In this program there are 14 tokens. These are *(,), +,* * and *digit* that contains ten tokens.

The institution is based on the fact that the parser does not care exactly which digit it sees, so as far as the parser is concerned, all digits are one and the same token: they form a token class [1].

It was said that the source type of the input can be string from console or file. The symbolic constants *ST_STRING* and *ST_FILE* are used for this reason respectively. For the *sourceType* field of the *SOURCE* structure it is initialized the appropriate one of these constants. Any token is represented by the *TOKEN* structure. If the type of the token is a digit, then the *type* field of this structure is *TT_DIGIT*, otherwise it is

the *ASCII code* of the token. The *token* field of the *TOKEN* structure is the representation of the token. *SOURCE* structure also contains union called *uBuf*. This is used with according to the *sourceType* field. *uBuf* union contains two fields, *f* and *pStr*. *f* used when the *sourceType* is *ST_FILE*, *pStr* is used when the *sourceType* is *ST_STRING*.

Furthermore, this file contains only one function prototype, *GetNextToken*. This function gets the next token according to the *sourceType* field of the *SOURCE* structure.

g_token global variable which has type of *TOKEN*, contains tokens. Since all the modules have to access this variable, it was defined global. Likewise, *g_source* global variable which has a type *SOURCE*, contains the source.

scanner.c file contains two static functions and one global function. Furthermore, this file contains two global variables, are *g_token* and *g_source*. *getNextTokenFromFile* and *getNextTokenFromString* functions are called from *GetNextToken* function according to the *sourceType* value of the *g_source* variable. These static functions get the next token and replace the *g_token* variable with the appropriate values.

2.4 Parsing or Syntax Analysis for the simple manual calculator

The parsing module of the simple manual calculator program is *parser.c* and its own header is *parser.h*. The contents of the *parser.h* file are shown in Appendix A.4 and the contents of the *parser.c* file are shown in Appendix A.5.

parser.h contains three symbolic constants that are used for the states of the return values of the functions. This file also contains two additional symbolic constants, *NT_DIGIT*, *NT_PARAN* for the type value of the *EXPRESSION* structure. *NT_DIGIT* is defined as *'D'* which is used for the digits. NT_PARAN is used for parenthesis and for operators. This file also contains function prototypes and declaration of the *EXPRESSION* structure.

parser.c file contains several static functions, *allocNode*, *freeNode*, *parseOperator,* and *parseExpression*. *allocNode* function allocates an *EXPRESSION* which is the node of the *AST* and *freeNode* function frees the *EXPRESSION* object. *parseOperator* is called for parsing the operators. *parseExpression* is a recursive function that produces the *AST* by calling the appropriate function. Furthermore, there are two global functions, *DisplayErrorMessage* and *Parse.* *DisplayErrorMessage* is called when any of the function returns error. *Parse* function is called for parsing the all source text.

AST for the expression *(2 * ((3 * 4) + 5))* is shown in Figure 2.7.

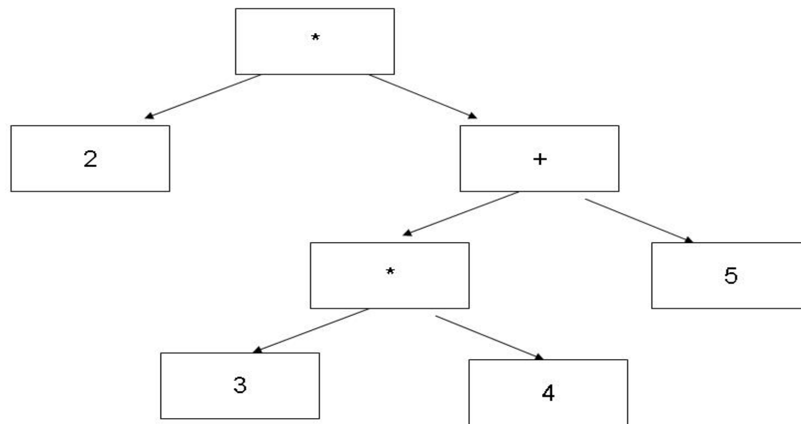


Figure 2.2 AST for the expression (2 * ((3 * 4) + 5))

2.5 Interpretation and code generation for the simple manual calculator

The Interpretation and code generation module of the simple manual calculator program is *backend.c,* and its own header is *backend.h*. The contents of the *backend.h* file are shown in Appendix A.6 and the contents of the *backend.c* file are shown in Appendix A.7.

backend.h file contains two function prototypes, *GenerateCode* and *Process.* *GenerateCode* function genetares the appropriate code for the machine. *Process* function interprets the code.

backend.c file contains two static functions, *generateCode* and *interpret.* *generateCode* function generates the appropriate machine code. Since this project is simple, *generateCode* only prints the machine codes. *interpret* function interprets the operation and returns the result. These functions are called from *GenerateCode* and *Process* respectively.

3. LEXICAL ANALYSIS OR SCANNING

The first process of the interpreters / compilers is the *lexical analysis* or *scanning*. *Atoms* or *terminal symbols* are used by the parser module. Several tools for scanning exists for example. Traditionally in UNIX systems there is a tool called *lex.* The GNU licensed version of the lex is called *flex.* For example in LINUX systems *lex* and *flex* are the same programs. The scanning module can also be written by the programmer. If it is written by the programmer, it will be particular for the language. It can be general only when the programmer writes a general tool like *lex or flex.*

3.1 Regular Expressions

Regular expression concept came from mathematics. *Regular expressions* are powerful language for describing and manipulating text [15]. In software, regular expressions are used to obtain the appropriate part of the string that ensures the given rule. Tools or programs that are used for this reason are called *regular expression engines*. There are several tools that use regular expressions. For example some editor programs can search by using regular expressions. *grep* command in UNIX/LINUX systems also use regular expressions for searching. Regular expression concept is abbreviated as *regex.*

In regular expressions, data that is appropriate for the given rule can be obtained. If we want to obtain the numbers in the text, we must use regular expressions. Also for example if we want to obtain date that has a format *dd/mm/yyyy*, we must use regular expression.

There are no standard in regex engines [5]. But they have few differences .In C and C++, there are no standard functions or classes that process the regular expressions;

however; in Java and .NET platforms standard classes about regular expressions exists.

Regular expressions are context-independent syntax that can represent a wide variety of character sets and character set orderings, where these character sets are interpreted according to the locale [14].

3.1.1 Using the Regular Expressions

The searching pattern is written by using some *meta characters.* If no meta character is used, exact match of the pattern is obtained.

'['and '] 'are meta characters. They represent a one character. For example, the for the pattern:

"h[ea]llo"

"hello" or "hallo" can be obtained. Meta characters are not normal character itself. If we want to use any meta character as a normal character, we must use '\' before the meta character. For example;

"h\[x\]a"

In this example '[' and ']' are not meta characters, they are normal. An interval can be pointed out by using '-' between '[' and ']'. For example;

"[0-9]"

In this example, the number between 0 and 9 can be obtained. Notice that again it represents only a one character. '-' is also a meta character. For example, the pattern above can be used to obtain a date that has a format dd/mm/yyyy.

"[0-9] [0-9]/ [0-9] [0-9]/ [0-9] [0-9] [0-9] [0-9]"

More than one interval can be written between '[' and ']'. For example;

"[0-9a-zx]"

In this form, a one character that can be between 0 and 9 or can be between 'a' and 'z' or only x can be obtained [5].

'^' is also a meta character and it means "not". For example;

"[^a]"

means any character that is "not" 'a' [6].

Some characters begining with '\' mean some group of characters. For example,

"\d" and "[0-9]", "\w" and "[a-z]", "\s" and "[\t\r\n]"

have the same meanings [5]. The capital letter version of these characters means "not". For example, "\D" and "[^0-9]" are similar.

‘.’ is a meta character and means *any character*. For example;

“[a-z]..b”

With this form a pattern - that has first character is between ‘a’ and ‘z’, then any two characters then character ‘b’ - can be obtained.

‘?’ is a meta character and means that left of this character is optional. For example,

“ab?c”

With this form, “ac” or “abc” can be obtained.

‘(‘ and ’)’ are also meta characters. They are used for grouping. For example;

“a(bc)?d”

In this example, “abcd” or “ad” can be obtained. ‘?’ can be used in ‘[]’. For example,

“a[0-9]?b”

With this form “ab” or “a-any number-b” can be obtained [6].

‘|’ is also a meta character and means “or”. For example;

“ab?c|a.b”

In this form, ‘|’ produces a two condition. “abc”, “ac”, “a-any character-b” can be obtained [5].

‘*’ and ‘+’ are important meta characters. They are very important. The little difference between there two characters is subtle. ‘*’ means, if there is zero or more left character of the ‘*’, all can be obtained. ‘+’ means, if there is one or more left character of the ‘+’, all can be obtained. For example;

“f+”

With this form, all contiguous ‘f’ characters can be obtained.

For a text ,

“yyyyyaaaacccc”

If we use “a*” form, we will get no character. Because when the search begins, zero character was found. If we use “a+”, ‘a’ character will found and all contiguous ‘a’ will be obtained.

“.*” or “.+” are frequently used. This form means “all character up to the end of line”. For example;

“okrn.*”

With this form, all characters from first occurrence of okrn to end of line can be obtained. For example;

“(abc)+”

With this form, contiguous “abc”s can be obtained.

There are of course a lot of meta characters and details.

3.1.2 Theory of Regular Expressions

Using regular expressions, we can specify patterns to lex that allow it to scan and match strings in the input. Each pattern in lex has an associated action. Typically an action returns a token, representing the matched string, for subsequent use by the parser.To begin with, however, it is simply printed the matched string rather than return a token value. It may be scanned for identifiers using the regular expression

letter(letter|digit)*

This pattern matches a string of characters that begins with a single letter, and is followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- repetition, expressed by the "*" operator

- alternation, expressed by the "|" operator

- concatenation

Any regular expression expressions may be expressed as a *finite state automaton* (FSA). FSA can be represented using states, and transitions between states. There is one start state, and one or more final or accepting states.

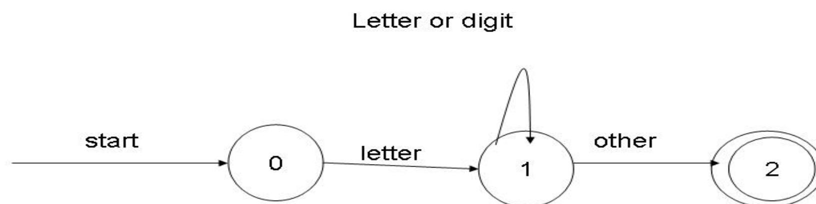


Figure 3.1 Finite State Automaton

In Figure 3.1, state 0 is the start state, and state 2 is the accepting state. As characters are read, Transition is made from one state to another. When the first letter is read, it is transit to state 1. When character is read other than a letter or digit, we transition to state 2, the accepting state. *Any*FSA may be expressed as a computer program. For example, 3-state machine is easily programmed [9]:

```
start: goto state0

state0: read c
        if c = letter goto state1
        goto state0

state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2

state2: accept string
```

3.1.3 Regular Expressions in C and C++

There are no standard functions or classes about regex in C and C++, however, there are several libraries written by anyone or company. There are POSIX functions about regex. These functions are standard for POSIX systems, UNIX, LINUX, MAC OS etc. Declarations of all these functions are in *regex.h* file. When these POSIX functions are used, fist *regcomp* function must be called with the parameter of the regular expression form. Then *regexec* function is called for searching. At last, *regfree* function called for changing the regex form. These functions interpret basic and extended Regular expressions [13]. A very simple use of these functions is shown in Appendix B.1.

3.2 Lex / Flex programs

lex file which is the input file for the lex/flex programs can be separated into three sections [7].

- definitions
- rules
- subroutines (user codes, C codes)

The format for this file is shown in Figure 3.1

Definitions
%%
Rules
%%
C codes

Figure 3.2 Format of the lex file [7].

Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system [11].

In this format %% characters distinguish the sections. *Definition section* can consist of macros and/or C declarations. C declarations are written between *%{* and *%}* like this:

```
%{
        int g_x;
        /*…*/
%}
```

Macros are the text representations of the regular expressions. For example,

```
DIGIT       [0-9]+
```

```
ALPHA       [a-zA-z]+
```

So, typical *definitions sections* can be like this [7]:

```
DIGIT       [0-9]+

ALPHA       [a-zA-Z]+

%{
        int g_x;
        /*…*/
%}
```

When macros are used in any other section they must be written between '{' and '}' like this:

```
{ALPHA}
```

It is not mandatory that lex file contains *definitions section*. C declarations in *definitions section* is exactly written to the code file that lex or flex program produces.

3.2.1 Rules Section

Every rule must be written in a line and the following way [7]:

```
<Pattern>     <action>
```

Pattern must be aligned to left. Action part consists of the C codes that are processed when the related atom found. If there are more than one C expression for the action, then the code must be written in the *{}*. For example;

```
DIGIT       [0-9]+

ALPHA       [a-zA-z]+


%{
        int g_digitNo = 0;
        int g_variableNo = 0;
%}

%%

{DIGIT}     {++g_digitNo;}
{ALPHA}     {++g_variableNo;}
```

If the there is no action for the pattern, the related item is wasted. If the *rules* section is empty, all characters are passed to stream.

3.2.2 Codes Section

Codes section consists of the C codes that are passed to the code file that the lex or flex programs produce. In this section there *main* or *yywrap* functions must be exist at least. If these are not written, and the *libfl.a* library is included the link phase, default version of these functions are written automatically. The default *main* function only calls the *yylex* function.

It is not mandatory that the code which is produced by *lex/flex* consists of the main function. In fact, the chief function is the *yylex* function. This function is defined by the lex/flex program. Also *yylex* function needs the yywrap function. yywrap function must be written by the programmer or must be declared as “not written“ via an exclusive directive [7]:

%option noyywrap

However, there is a *yywrap* function defined as follows [7];

```
int yywrap(void)
{
        return 1;
}
```

yylex is the chief function of the *lex/flex*. If the programmer uses the *return* in the *action part,* the code returns from the *yylex* function.

3.2.3 Important Global Variables in the lex/flex Source Code

yytext variable consists of the current atom. *yyleng* consists of the length of the current character.

Two significant global variables are *yyin* and *yyout* that has a type *FILE *. Lex/flex* program use the file that *yyin* points to as a source (input) file [7]. The default file is the *stdin* for *yyin.* . Lex/flex program use the file that *yyout* points to as a destination (output) file. The default file is the *stdout* for *yyout* [7].

3.2.4 Compiling and Executing the lex/flex source file

Lex/flex source file is not a C code file. *Lex/flex* writes a C code that tokenizes. As a result of this process, *lex.yy.c* file is produced. This file is C code file. This file must be compiled by gcc. These processes are;

1. *Lex/flex* source file is written with an extension *.lex* or *.flex* or *.l*.
2. *Lex/flex* source file is processed with the *lex/flex* as:

    flex test.c

Afterwards *lex.yy.c* file is produced.

3. *lex.yy.c* file is compiled with gcc;

   gcc –o test lex.yy.c –lfl

3.2.5 Lexical Analysis Process with yylex Function

yylex function that is written by *lex/flex* behaves as:

1. Takes the characters from the file that the *yyin* points to. Then analyze them.
2. If the characters that are taken from the source file (*yyin*) ensures the *regex* rule, they are copied to the array which is pointed to by *yytext*. Then the action related to the *regex* rule is processed.
3. If the characters that are taken from the source file (*yyin*) does not ensure the *regex* rule, these characters are passed to the *yyout* file.

yylex function conserves the state. In other words, if *yylex* function is terminated by *return expression*, when it is called again it continues from before state.

3.2.6 Using lex/flex in Projects

The aim of using the *lex/flex* is to leave the *scanning process* to this program. *yylex* function must be called for each atom. A typical arrangement must be as follows:

1. In action parts of the *lex/flex* source file the programmer must use *return* for terminating the *yylex* function. Programmer takes the atom from *yytext* after terminating *yylex*.

2. *yylex* function is returned with the type of the atom. So that, it can be possible to establish the type of the atom.
3. At the end of the file, *yylex* function calls *yywrap* automatically then return with 0.

3.3 Manual Lexical Analysis

Lexical Analysis can be done without using any tool (manually, programmatically). Lexical Analysis can be processed as follows:

1. *White spaces* are skipped. The first character that is not a white space is taken.
2. This character is related to the atoms type whether it is numerical or alphanumerical etc.
3. Contiguous characters are taken with respect to this character.

3.4 Object Oriented Manual Lexical Analysis by using C++

Lexical Analysis process can be implemented manually by using C++ and its *classes*. This method is less effective because of the level of C++ programming Language. The class diagram of this design is shown in Figure 3.2.

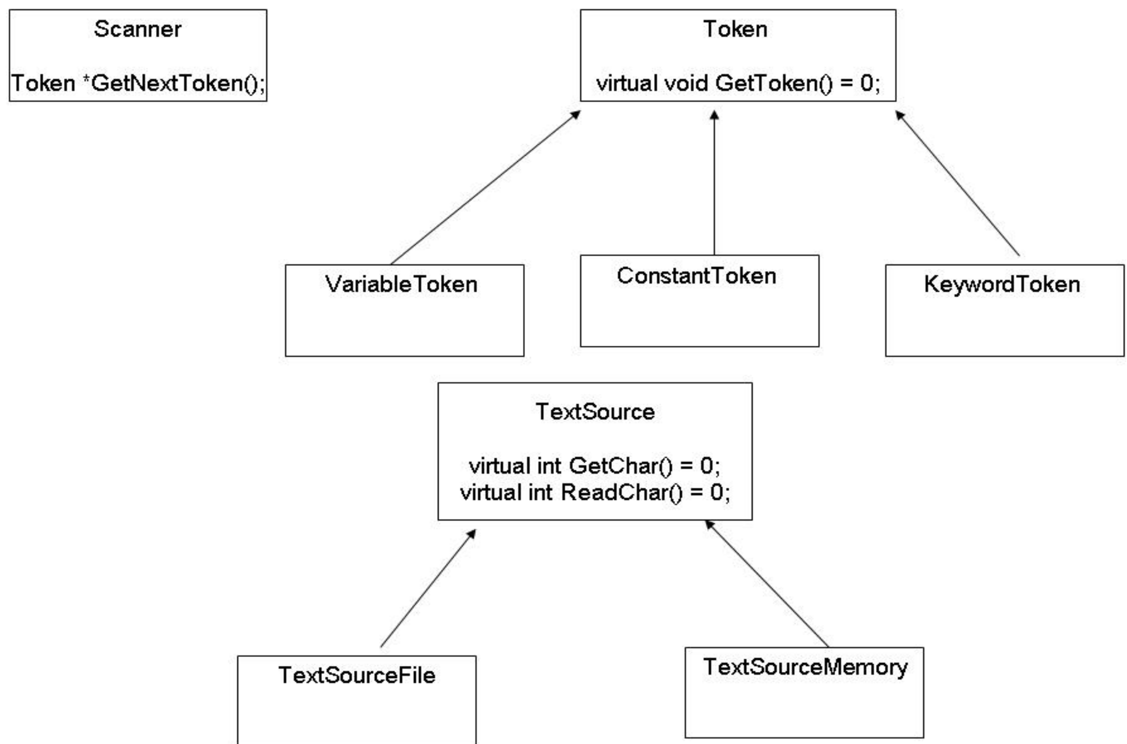


Figure 3.3 Class diagram of Object Oriented Lexical Analysis Process

Scanner class tokenizes the atoms. As the main class for lexical analysis. *Tokens* are represented by classes that are inherited from *Token* class. For example, *VariableToken class* represents the variables, whereas *Token* class has a *GetToken* function. This function is *pure virtual*. This function is overridden in derived classes. *Token class* is used by *Scanner class* with the *association technique* in *Object Oriented Paradigm.* The main function that gets the atom is the *GetNextToken* member function of the *Scanner* class. *GetNextToken* first skips the *white space characters.* It gets the first character except *white space,* creates the appropriate *token object*. Then it calls the *GetToken* function of the *token object.*

Input source can be *file* or *memory. TextSource class* is the base of the *TextSourceFile* and *TextSourceMemory classes.* The former class represents the file input, latter represents the memory input. *TextSource class* has two *pure virtual*

functions which are named as *GetChar* and *ReadChar.* These functions are overridden in derived classes.

Dynamic Allocation for any *token variable* is not effective. For this reason, pre-created token variables can be used.

The implementation file *scanner.h* and scanner.cpp are shown in Appendix B.2 and Appendix B.3 respectively.

scanner.h file consists of the declarations of the classes which are shown in Figure 3.2 and any *ScannerException class* that is the exception class of the *Scanner* class. *ScannerException class* is used for *exception handling. Also, scanner.c* file consists of the implementations of the classes which are shown in Figure 3.2 and any *ScannerException class* that is the exception class of the *Scanner* class.

4. SYNTAX ANALYSIS OR PARSING

The other process of the *interpreter / compiler* is the *syntax analysis* or *parsing*. Several tools for *parsing* exist. Traditionally in *UNIX* systems there is a tool called *yacc*. The *GNU* licensed version of the *yacc* is called *bison*. *bison* is also the extended version of the *yacc*. The *parsing module* also can be written by the programmer. If it is written by the programmer, it will be particular for the language. It can be general only when the programmer writes a general tool like *yacc or bison.*

In the syntax-analysis phase, a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. If the tokens in a string are grouped according to the language's rules of syntax, then the string of tokens generated by the lexical analyzer is accepted as a valid construct of the language; otherwise, an error handler is called [12].

4.1 yacc/bison Programs

yacc and *bison* are programs that make *syntax analysis/parsing* by using *BNF* notation. *yacc* and *bison* are compatible with *lex* and *flex*. *Bison* is the extended and *GNU* version of *yacc*. There is an open source version of the *bison* for *Linux* and *Windows* operating systems.

4.1.1 Organization of yacc/bison source (input) file

yacc/bison source file can be separated into three sections [8].

- Definitions
- Rules
- Subroutines (user codes, C codes)

The format for this file is shown in Figure 4.1

Definitions
%%
Rules
%%
C codes

Figure 4.1 Format of the lex file [8].

Definitions section can consists of the exclusive declarations of *yacc/bison* or *C* declarations. C declarations must be written between *%{* and *%}. Rules section* consists of the *BNF productions.* All *productions* must be terminated with *';'* character. *Code of productions (action)* can be defined for all options that separated by '/' [8].

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. [10] .

A sample *rules section* can be as follows:

```
expression:
        expression '+' factor  {$$ = $1 + $3;}
        | expression '-'factor  {$$ = $1 - $3;}
;
factor:
        factor '*' NUMBER  {$$ = $1 * $3;}
        | NUMBER           {$$ = $1;}
;
```

If the terminal symbol includes one characters, it is written in ' and '. Otherwise, it must be declared with *%token* in the *declarations section.* For the example, above +, -, * and *NUMBER* are atoms. Derivations for this example are;

*NUMBER * NUMBER + NUMBER * NUMBER*

or

NUMBER

For productions, *$1* represents the first symbol, *$2* represents the second symbol etc. *$$* represents the result of production.

4.1.2 Process of yacc/bison

When the source code of the *yacc/bison* is compiled, if the source file's name is *test.y*, *test.tab.c* C code file is generated. In this C code file parsing processes are done with the *yyparse* function, therefore a programmer must write a *main* function and must call *yyparse* to excute that code. Algorithm of the yyparse function is written as follows:

1. *yyparse* needs atoms for *root production*. It gets the atoms and derives the production
2. *yyparse* gets atoms by using *yylex.* For that reason, *yylex* function must be written by programmer or by *lex/flex.*
3. *yyparse* calls the *yylex* function then retrieves the numeric value of the atom from *yyval* variable. If *lex/flex* is used, programmer must assign the value to the *yylval* variable before *yylex* returns.
4. *yylex* function establishes the type of the atom from *yylex* function. If the return value of the *yylex* function is between 0 and 255, atom includes one character. The return value is the *ASCII code* of this character. If this value is greater than 255 it is a multi-character atom. The type of the atom can be established by this value [10].

5. When *yacc/bison* programs are executed with *–d* command line, it produces also a *.tab.h* header file; *test.tab.h i.e.* This file includes the symbolic constants that are declared as *%token.* For this reason, these constants must be returned from *yylex* by programmer. Therefore, this file must be *included* in the *lex/flex* source file.

4.1.3 Expression and Atom Types in yacc/bison

The types of *yylval* that consists of found atom, *$n* that consists of the related *non-terminal* and *terminal symbols* is*YYSTYPE.* If *YYSTYPE* is not defined with *#define* declaration in the generated C code file, *YYSTYPE* is defined as int by yacc/bison. If the programmer wants the *YYSTYPE* as double for example, he/she have to define *YYSTYPE* as double in the *definitions section* as follows:

```
%{
        #define YYSTYPE    double
%}
```

It is possible to differentiate the type of atoms by using *%union* declaration. A sample *%union* declaration is shown in Figure 4.2;

```
%union {
        double val;
        char *name;
}
```

Figure 4.2 Sample %union declaration.

It is also possible to declare type for *non-terminal and terminal symbols* by using *%type and %token* declarations, respectively. Types are written in < *and* >. For example;

```
%token <val> NUMBER
%token <name> NAME
%type <val> expression
```

With the *%union* declaration type of the *yyval* becomes a union as shown in Figure 4.2. When we use *yacc/bison* with *–d* command option, *YYSTYPE* is defined with *#define declaration* as union that is declared with *%union* in source file of the *yacc/bison* program. Also an *extern declaration* is declared in the generated header file. Briefly,

1. Type of the *yyval* is always *YYSTYPE.*
2. Default type of the *YYSTYPE* is *int*
3. If *YYSTYPE* is defined with *#define, YYSTYPE* become type as defined
4. If *YYSTYPE* is defined as *%union* declaration, *YYSTYPE* become this *union* type.

4.1.4 Using C++ codes with *lex/flex* and *yacc/bison*

Lex/flex and yacc/bison program generates C codes. If the extensions of these files are renamed as *.cpp* then are compiled by C++ compiler, C and C++ compatibility problems can be occurred. Therefore, these codes must be compiled with C compiler, and other C++ codes must be compiled with C++ compiler. Then all the *object codes* must be *linked* together, however, the following problems can be occurred:

1. *Name decoration* of the compilers can be different. Therefore, C and C++ codes must be compiled by the compiler that belongs to the same *compiler family.*
2. *C++* functions are called by the *C modules* must be declared and defined with "*extern C" declaration.*

3. For *dynamic memory allocation* in C++ *operator new* and *operator delete* must call *malloc* and *free* functions respectively. Therefore, *operator new* and *operator delete* must be implemented (defined) again.

4.2 Representing the Syntax Tree by Using C

Syntax Tree must be expressed as a data structure by using *C*. It is aimed to be expressed as a data structure in memory not to draw the figure of the tree. For the Figure 1.2, *nodes* may have various elements. For effective and comfortable usage, nodes must be represented homogenously. One field of the structure of the node can be used for determining the type of the node. For example, node belongs to binary or unary operator. Furthermore, node belongs to a non-terminal symbol. If the node belongs to an operator, all operands will be *pointers* that have the same types. A sample *node structure* is shown in Figure 4.3:

```
typedef struct _tagNODE {
        int nodeType;
        union {
                OPERATOR_NODE opNode;
                VARIABLE_NODE opNode;
                CONSTANT_NODE constNode;
        } type;
} NODE, *PNODE;;
```

Figure 4.3 Sample node structure

The *nodeType* field of the structure represents the active field in the *type* union. n*odeType* can be the *enumeration* that can be declared as follows:

```
enum {
        NT_OPERATOR,
        NT_VARIABLE,
        NT_CONSTANT
};
```

There are some unnecessary nodes in the tree that is shown in Figure 1.2. For example; *additive_expression* below the *expression* is unnecessary. No element exists for these unnecessary nodes in the *NODE* structure.

4.3 Constructing a Syntax Tree by using yacc/bison

Syntax Tree for expressions can be constructed easily by using *yacc/bison*. Syntax Tree is constructed by allocating nodes in *productions*. For example, *definitions sections* of the *yacc/bison* source file for a calculator that supports +, -, *, / is written as follows:

```
expression:
        additive                {$$ = processExp($1);  /*Code 1*/}
;
additive:
        additive '+' factor     {$$ = processBinaryOp(OP_ADD, $1, $3); /*Code
2*/}
        | additive '-' factor   {$$ = processBinaryOp(OP_SUB, $1, $3); /*Code 3*/}
        | factor                {$$ = $1; /*Code 4*/}
;
factor:
        factor '*' NUMBER       {$$ = processBinaryOp(OP_MUL, $1, $3); /*Code
5*/}
        | factor '/' NUMBER     {$$ = processBinaryOp(OP_DIV, $1, $3); /*Code 6*/}
        | NUMBER                {$$ = processNumber($1); /*Code 7*/}
;
```

yacc/bison executes the C codes for *productions* from bottom to top. For example, for the derivation below for the above example:

*NUMBER * NUMBER + NUMBER + NUMBER*

The C codes are executed as follows:

1. This is an *expression* and this is also an *additive*. The C code in the *additive* excuted at last.
2. C codes for the NUMBER are executed first.

Process Order of *NUMBER * NUMBER + NUMBER + NUMBER* is as follows*:*

1. Code 7
2. Code 7
3. Code 7
4. Code 7
5. Code 5
6. Code 5
7. Code 2
8. Code 1

To construct syntax tree, all the codes above allocates a *NODE* structure. For example, C*ode 7* can allocate a *NODE* structure for *NUMBER* and can assign the related number. *factor* can get the two *NODE* structures represents two *NUMBER* atoms (terminal symbol), then allocates another *NODE* structure. *additive can* get the two *NODE* structures represents two *factor* non-terminal symbol, then allocates another *NODE.* When expression is encountered, *syntax tree* is being constructed. To calculate the related expression, a recursive function that gets a *NODE* as parameter can be implemented. In this method, the result is calculated at the top of the

productions. From bottom to top, data of the symbols are recorded and the *syntax tree* is constructed.

4.4 Error Handling in yacc/bison

A nice compiler gives the user meaningful error messages [9]. When the error is encountered, default behavior of the *yacc/bison* is to terminate the program. *yyparse* function calls *yyerror* by default. *yyerror* is compatible with *lex/flex*. *yyparse* function terminates the program after *yyerror* returns. For effective error handling the built in *error* non-terminal symbol is used. This symbol is processed as a normal non-terminal symbol. For example:

```
statement:
        null_statement          /**/
        | if_statement  { $$ = processStatement($1, STM_IF); }
        | error         { /*Error Code*/ };
;
```

4.5 Manual Syntax Analysis

Syntax Anaysis can be done without using any tool. Of course, for syntax analysis the module that gives the next *atom* can be used as input. Indeed, *yacc/bison* call the *yylex* function for getting the next atom from the output of the lex*/flex* program.

Some typical algorithms can be used for *syntax analysis.* The popular algorithm is to process the infix operators as postfix via a *stack.* In this method, the opearation with two operands processes as follows:

1. Left operand is pushed to stack
2. Right operand is pushed to stack

3. Two of these operands are popped from the stack and processed
4. Result is pushed to stack

4.6 Object Oriented Manual Syntax Analysis by using C++

Syntax Analysis process can be implemented manually by using C++. This method is less effective because of the level of C++ programming Language.

For syntax analysis, *Parser* class can be implemented that use the *Scanner* class as a variable that is written in the previous chapter. *parser.cpp* file that contains the implementation of *Parse* class is shown in Appendix C.2, and also its header file *parse.h* is shown in Appendix C.1.

This skeletal *Parser* class is implemented in *parser.cpp* file and declared in *parser.h* file. *getNextToken* function gets the next atom by using *Scanner class* object. *m_pToken* member variable contains the atom and *m_pTokenType* variable contains the type of the atom. Parse operation is started by calling the function that parse the least precedence operator. This function is named as *doAdditiveExpression.* All functions that parse operators call the function that belongs to higher precedence operator. All results are pushed to stack. The atom is get by *getNextToken* function when the type of the atom is established.

5. SAMPLE INTERPRETER

This sample interpreter is the detailed skeletal code of interpreter. Interpreter supports loops, *if statements*, several various *operators* etc. With this code, advanced compiler or interpreter can be improved. This is not the introduction to the advanced compiler. This is big part of the advanced compiler / interpreter.

5.1 Design of Sample Interpreter

In this project, all *terminal symbols* and *non-terminal symbols* represented as *NODE* structure that is declared as follows:

```
typedef struct tagNODE {
        int nodeType;
        union {
                NODE_CONSTANT nodeConstant;
                NODE_VARIABLE nodeVariable;
                NODE_OPERATOR nodeOperator;
                NODE_STATEMENT nodeStatement;
                NODE_STATEMENT_IF nodeStatementIf;
                NODE_STATEMENT_WHILE nodeStatementWhile;
                /*…*/
        } type;
} NODE, *PNODE;
```

Parse Tree is constructed with this *NODE* structure. After that, all operations are done by using this tree with a recursive function. Any node in the *Parse Tree* belongs to a different *terminal symbol* or *non-terminal symbol* but all the nodes are represented as a *NODE* structure. *NODE* structure can consist of constant, variable, operator or statement. *Constants* can be represented as *NODE_CONSTANT* structure. This structure consists of the value of the constant. *NODE_CONSTANT* structure is declared as follows:

```
typedef struct tagNODE_CONSTANT {
        double value;
} NODE_CONSTANT, *PNODE_CONSTANT;
```

Names, values and types of the variables are saved in the *symbol table*. N*ODE_VARIABLE* consists of the index of the variable in the *symbol table. NODE_ VARIABLE* structure is declared as follows:

```
typedef struct tagNODE_VARIABLE {
        int index;
} NODE_ VARIABLE, *PNODE_ VARIABLE;
```

The structure that represents the operator which is named as *NODE_OPERATOR* consists of the operator type and the operands. Operands of the operators are represented as *NODE* structure. For operands array with three elements is allocated. All the slots of this array is not always used. They are used according to the type of the operator. *NODE_OPERATOR* structure is declared as follows:

```
typedef struct tagNODE_OPERATOR {
        int opType
        PNODE operands[3];
} NODE_OPERATOR, *PNODE_OPERATOR;
```

Any *statement* is represented as *NODE_STATEMENT* structure. *NODE_STATEMENT* structure is declared as follows:

```
typedef struct tagNODE_STATEMENT {
        int stmType;
        int stmInfo;
        PNODE pNode;
} NODE_STATEMENT, *PNODE_STATEMENT;
```

Some statements are represented as exclusive structures. For example, *if statements* are represented as *NODE_STATEMENT_IF, while statements* are represented as *NODE_STATEMENT_WHILE etc. NODE_STATEMENT_IF* and *NODE_STATEMENT_WHILE* structures are declared as follows:

```
typedef struct tagNODE_STATEMENT_IF {
        PNODE pNodeExpression;
        PNODE pNodeTrue;
        PNODE pNodeFalse;
} NODE_STATEMENT_IF, *PNODE_STATEMENT_IF;

typedef struct tagNODE_STATEMENT_WHILE {
        PNODE pNodeExpression;
        PNODE pNodeTrue;
} NODE_STATEMENT_WHILE, *PNODE_STATEMENT_WHILE;
```

The main grammar structure of the sample interpreter is shown in Figure 5.1:

```
input:
        | input_statement
;
statement:
        null_statment
        | simple_statement
        | compound_statement
        | command_statement
        | if_statement
        | while_statement
;
null_statment:
        ‘;’
;
simple_statement:
        expression ‘;’
;
compound_statement:
        ‘{’ ’}’
        | ‘{’ statement_list ’}’
;
statement_list:
        statement statement_list
        | statement
;
command_statement:
        print_command_statement:
        | exit_command_statement
;
```

Figure 5.1 Grammar structure of the sample interpreter

```
print_command_statement:
        TOKEN_PRINT_COMMAND expression ';'
;
exit_command_statement:
        TOKEN_EXIT_COMMAND ';'
;
if_statement:
        TOKEN_IF_STATEMENT '(' expression ')' statement
        | TOKEN_IF_STATEMENT '(' expression ')' statement
                        TOKEN_ELSE_STATEMENT statement
;
while_statement:
        TOKEN_WHILE_STATEMENT '(' expression ')' statement
;
expression:
        assignment_expression
;

/*…*/
```

Figure 5.1 Grammar structure of the sample interpreter (continued)

Types of n*on_terminal symbols* are represented as *NODE * (PNODE)* in *yacc/bison* source code file. *Number or variable* are represented as *NODE.* For example, the syntax tree of a statement

*a = b + 10 * 20;*

is shown in Figure 5.2

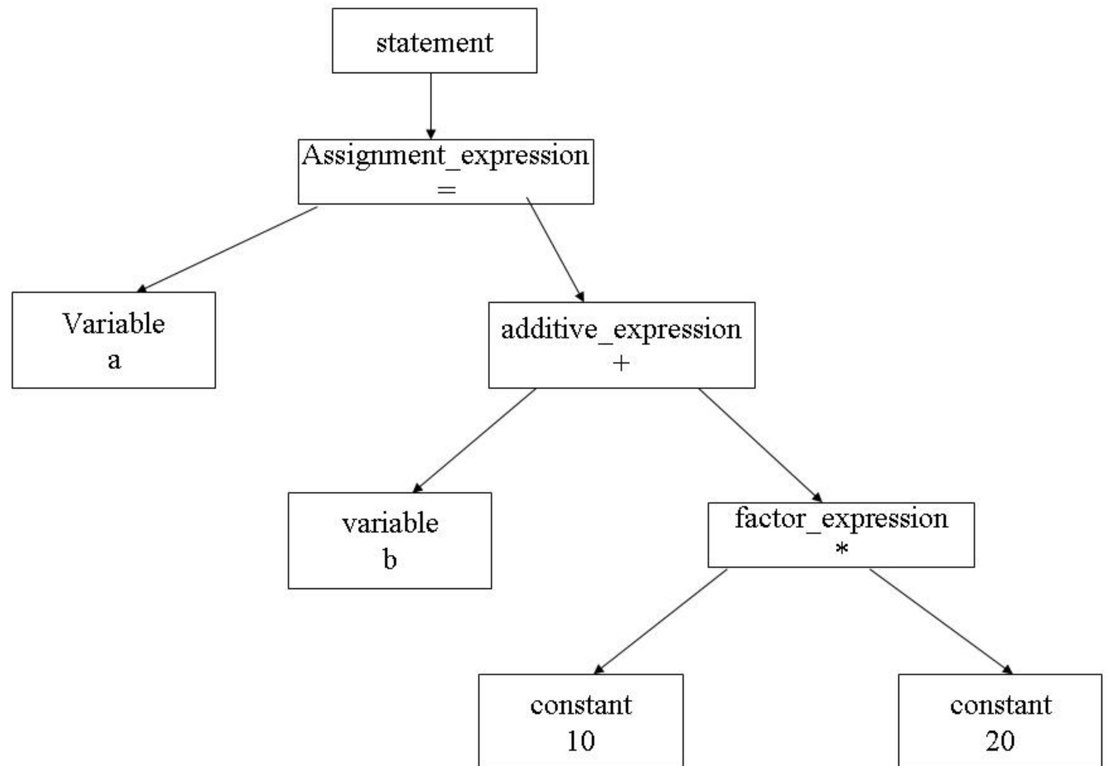


Figure 5.2 Syntax tree of *a = b + 10 * 20;*

The most important function of this interpreter code is the *execute* function. This function fetches the first node of the syntax tree as a parameter and calculates the statement in a recursive manner. Prototype of this function is as follows:

```
double execute(PNODE pNode);
```

Another important function is *freeNode* function. This function frees the tree. Prototype of this function is as follows:

```
static void freeNode(PNODE pNode);
```

5.2 Implementation of Sample Interpreter

Sample interpreter project consists of several code files. *Lex/flex* source file is named as *interpreter.l*. This file is used for acquiring the atoms. i*nterpreter.l* file is shown in Appendix D.1.

yacc/bison code file is named as *interpreter.y.* This file is used for syntax analysis. This file is shown in Appendix D.2.

When *lex/flex and yacc/bison programs are used for interpreter.l and interprter.y* files respectively. *Interpreter.tab.c and lex.yy.c* files are generated respectively. Furthermore, *interpreter.tab.h* file is also generated. These files are compiled and linked together with other files.

interpreter.h file contains thedeclarations of essential structures for the grammar. For example: *NODE, NODE_VARIABLE etc.* This file also contains the essential *enumerations* and function prototypes. This file is included by *both interpreter.l and interpreter.y.* Also, automatically this file is included by *interpreter.tab.c and lex.yy.c. interpreter.h file is* shown in Appendix D.3:

interpreter.cpp file contains the implementation of *symbol table* data structure. *interpreter.hpp* file contains the declaration of this data structure. *interpreter.cpp* and *interpreter.hpp* files are shown in Appendix D.4 and Appendix D.5 respectively.

5.3 Properties of Sample Interpreter

Sample interpreter supports the *C programming syntax* but not all of them. For example, Control statements like *if statement* is supported. Loop statements of the *C*

are supported by Sample Interpreter etc. A sample code belongs to Sample Interpreter is as follows:

```
x = 12;

iken (x >0) {
        eger (x % 2 == 0)
        yazdir x;
        x = x – 1;
}
```

REFERENCES

[1] Grune, D., Bal, H., E., Jacobs, C. and J., H., Langendoen, K., G. *Modern Compiler Design:* John Wiley & Sons, Ltd, New York, (2003)

[2] Mak, R. *Writing Compilers and Interpreters: An Applied Approach Using C++*. John Wiley & Sons, Ltd, United States of America, Second Edition, (1996)

[3] Muchnick, S., S. *Advanced Compiler Design and Implementation:* Academic Press, United States of America, (1997).

[4] Aho, A., V, and Ullman, J., D. *Compilers: Principles of Compiler Design*: Addison_Wesley, Massachusetts., (1977)

[5] Friedl J., E., F. *Mastering Regular Expressions:* O'Reilly & Associates, Cambridge, Second Edition (2003)

[6] Stubblebine T.*Regular Expression*: Pocket Refence. O'Reilly & Associates, Cambridge, (2003)

[7] Paxon V. *FLEX(1):* Internet WWW-page, URL: http://www.hmug.org/man/1/lex.html (24.04.2005).

[8] Donnelly C., Stallman R., M. *Bison Manual for Version 1.875:* Free Software Foundation, USA, 8th edition (2003).

[9] Niemann T., *A Compact Guide to Lex&Yacc:* Internet WWW-page, URL: www.epapers.com, (20.4.2005)

[10] Johnson, C., J., *Yacc: Yet Another Compiler-Compiler:* Internet WWW-page, URL: http://www.cs.rpi.edu/~moorthy/Courses/compilerf05/yacc.pdf (19.05.2005)

[11] Lesk M., E., and Schmidt E. *Lex – A Lexical Analyzer Generator:* Internet WWW-page, URL: http://wolfram.schneider.org/bsd/7thEdManVol2/lex/lex.pdf (19.05.2005)

[12] Kakde O., D. *Algorithms for Compiler Design:* Laxmi Publication, Massachusetts, (2003)

[13] *Standard for Information Technology—Portable Operating System Interface (POSIX)*
System Interfaces, Issue 6. Institute of Electrical and Electronics Engineers, Inc. and The Open Group, USA (2001)

[14] *Standard for Information Technology—Portable Operating System Interface (POSIX)*
Base Definitions, Issue 6. Institute of Electrical and Electronics Engineers, Inc. and The Open Group, USA (2001)

[15] Liberty J., *Programming C#:* O'Reilly & Associates, Cambridge, Second Edition (2002)

[16] Nilges E., G. *Build Your Own .NET Language and Compiler:* A Press, USA, (2004)

[17] Benders, F., J. ,F., Haaring, J., W., Janssen, T., H., Meffert, D. and Oostenrijk, A., C. *Compiler Construction : A Practical Approach*., University of Arnhem and Nijmegen, (2003)

[18] Aaby, A, A, *Compiler Construction using Flex and Bison:* Internet WWW-page, URL: http://cs.wwc.edu/~aabyan/Linux/compiler.pdf *(10.10.2005).*

[19] Levine J., R., Mason T., and Brown D., *lex & yacc:* O 'Reilly and Associates, Inc, United States of America, (1992).

[20] Sahni S., Data Structures, *Algorithms, and Applications in C++:* McGraw-Hill, USA, (1998).

[21] Terry P., D., *Compilers and Compiler Generators an introduction with C++*: International Thomson Computer Press, USA, Book & Disk Edition1997

6.1 APPENDIX A

6.1.1. main.c file

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "scanner.h"
#include "parser.h"
#include "backend.h"

int main(void)
{
      PNODE pNode;

      g_source.sourceType = ST_FILE;
      if ((g_source.uBuf.f = fopen("deneme.c", "r")) == NULL) {
            fprintf(stderr, "Cannot open file..\n");
            exit(EXIT_FAILURE);
      }

      if (Parse(&pNode) != SUCCESS)
            DisplayErrorMessage("Error in Expression");

      Process(pNode);

      return 0;
}
```

6.1.2. scanner.h file

```
#ifndef _SCANNER_H_
#define _SCANNER_H_

/*Include Files*/

#include <stdio.h>

/* Symbolic Constants*/

#define ST_FILE         1
#define     ST_STRING   2

#define TT_DIGIT  257
#define TT_EOF          256
```

```
/*Type definitons*/

typedef struct tagSOURCE { /*Source Structure*/
	int sourceType;
	union {
		FILE *f;
		char *pStr;
	} uBuf;
} SOURCE, *PSOURCE;

typedef struct tagTOKEN { /*Token structure*/
	int type;
	char token;
} TOKEN;

/*Function Prototypes*/

void GetNextToken(void);

/*extern declerations*/

extern TOKEN g_token;
extern SOURCE g_source;

#endif
```

6.1.3. scanner.c file

```
#include <stdio.h>
#include "scanner.h"

/*Static Function Prototypes*/

static void getNextTokenFromFile(void);
static void getNextTokenFromString(void);

TOKEN g_token;
SOURCE g_source;

static void getNextTokenFromFile(void)
{
	int ch;
	static long curOffset = 0;

	fseek(g_source.uBuf.f, curOffset, SEEK_SET);

	while ((ch = fgetc(g_source.uBuf.f)) != EOF)
		if (ch != ' ' && ch != '\n' && ch != '\t')
			break;

	if (ch == EOF) {
		g_token.type = TT_EOF;
		g_token.token = '#';

		return;
	}
```

```
	if (ch >= '0' && ch <= '9')
		g_token.type = TT_DIGIT;
	else
		g_token.type = ch;

	g_token.token = ch;
	curOffset = ftell(g_source.uBuf.f);
}

static void getNextTokenFromString(void)
{
	static int offset = 0;

	/*Skip white spaces*/
	for (; g_source.uBuf.pStr[offset] != '\0'; ++offset)
		if (g_source.uBuf.pStr[offset] != ' ' &&
g_source.uBuf.pStr[offset] != '\n' && g_source.uBuf.pStr[offset] !=
'\t')
			break;

	/*EOF encountered*/
	if (g_source.uBuf.pStr[offset] == '\0') {
		g_token.type = TT_EOF;
		g_token.token = '#';

		return;
	}

	if (g_source.uBuf.pStr[offset] >= '0' &&
g_source.uBuf.pStr[offset] <= '9')
		g_token.type = TT_DIGIT;
	else
		g_token.type = g_source.uBuf.pStr[offset];

	g_token.token = g_source.uBuf.pStr[offset];
	offset++;
}

void GetNextToken(void)
{
	static int offset = 0;

	if (g_source.sourceType == ST_FILE)
		getNextTokenFromFile();
	else
		getNextTokenFromString();
}
```

6.1.4. parser.h file

```
#ifndef _PARSER_H_
#define _PARSER_H_

/*Symbolic Constants*/

#define SUCCESS              1
#define FAILED               0
#define ERROR                -1

#define NT_DIGIT        'D'
#define NT_PARAN        'P'

/*Type definitions*/

typedef struct tagEXPRESSION { /*Expression structure*/
      int type;
      int val;
      int op;
      struct tagEXPRESSION *pLeft, *pRight;
} EXPRESSION, *PEXPRESSION, NODE, *PNODE;

/*Function Prototypes*/

void DisplayErrorMessage(const char *str);
int Parse(PNODE *ppNode);


#endif
```

6.1.5. parser.c file

```
#include <stdio.h>
#include <stdlib.h>
#include "parser.h"
#include "scanner.h"

/*Static Function Prototypes*/

static PEXPRESSION allocNode(void);
static void freeNode(PEXPRESSION pExp);
static int parseOperator(int *pOp);
static int parseExpression(EXPRESSION **ppExp);

/*Function Definitions*/

static PEXPRESSION allocNode(void)
{
      return (PEXPRESSION) malloc(sizeof(EXPRESSION));
}

static void freeNode(PEXPRESSION pExp)
{
      free(pExp);
}
```

```
static int parseOperator(int *pOp)
{
	if (g_token.type == '+') {
		*pOp = '+';
		GetNextToken();

		return SUCCESS;
	}
	if (g_token.type == '*') {
		*pOp = '*';
		GetNextToken();

		return SUCCESS;
	}

	return FAILED;
}

static int parseExpression(PEXPRESSION *ppExp)
{
	PEXPRESSION pExp;

	if ((pExp = allocNode()) == NULL)
		return ERROR;

	*ppExp = pExp;

	if (g_token.type == TT_DIGIT) {
		pExp->type = NT_DIGIT;
		pExp->val = g_token.token - '0';
		GetNextToken();

		return SUCCESS;
	}

	if (g_token.type == '(') {
		pExp->type = NT_PARAN;
		GetNextToken();
		if (parseExpression(&pExp->pLeft) == FAILED)
			DisplayErrorMessage("Missing Expression");

		if (parseOperator(&pExp->op) == FAILED)
			DisplayErrorMessage("Missing Operator");

		if (parseExpression(&pExp->pRight) == FAILED)
			DisplayErrorMessage("Missing Expression");

		if (g_token.type != ')')
			DisplayErrorMessage("Missing )");

		GetNextToken();

		return SUCCESS;
	}

	freeNode(pExp);

	return FAILED;
}

void DisplayErrorMessage(const char *str)
```

```
{
	fprintf(stderr, "%s\n", str);
}

int Parse(PNODE *ppNode)
{
	PEXPRESSION pExp;

	GetNextToken();
	if (parseExpression(&pExp) == SUCCESS) {
		if (g_token.type != TT_EOF)
			DisplayErrorMessage("Garbage After end of
program");

		*ppNode = pExp;

		return SUCCESS;
	}

	return FAILED;
}
```

6.1.6. backend.h file

```
#ifndef _BACKEND_H_
#define _BACKEND_H_

/*Function Prototypes*/

void GenerateCode(const PNODE pNode);
void Process(const PNODE pNode);


#endif
```

6.1.7. backend.c file

```
#include <stdio.h>
#include "parser.h"
#include "backend.h"

/*Static Function Prototypes*/

static void generateCode(const PEXPRESSION pExp);
static int interpret(const PEXPRESSION pExp);

static void generateCode(const PEXPRESSION pExp)
{
	switch (pExp->type) {
		case NT_DIGIT:
			printf("PUSH %d\n", pExp->val);
			break;
		case NT_PARAN:
			generateCode(pExp->pLeft);
			generateCode(pExp->pRight);
			switch (pExp->op) {
```

```
                        case '+':
                              printf("ADD\n");

                              break;
                        case '*':
                              printf("ADD\n");
                              break;
                  }
                  break;
      }
}

static int interpret(const PEXPRESSION pExp)
{
      switch (pExp->type) {
            case NT_DIGIT:
                  return pExp->val;
            case NT_PARAN:
                  {
                        int leftExp = interpret(pExp->pLeft);
                        int rightExp = interpret(pExp->pRight);

                        switch (pExp->op) {
                              case '+':
                                    return leftExp + rightExp;

                              case '*':
                                    return leftExp * rightExp;

                        }
                  }
                  break;
      }

      return 0;
}

void GenerateCode(const PNODE pNode)
{
      generateCode(pNode);
}

void Process(const PNODE pNode)
{
      printf("%d\n", interpret(pNode));
}
```

6.2 APPENDIX B

6.2.1. Simple use of POSIX regex functions

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <regex.h>

int main(int argc, char *argv[])
{
	FILE *f;
	long fileSize;
	char *pString;
	regex_t reg;
	regmatch_t pMatch[10];
	int i, j;

	if (argc != 3) {
		fprintf(stderr, "Wrong number of arguments\nUsage:
sampleregex filename regex");
		exit(EXIT_FAILURE);
	}

	if ((f = fopen(argv[1], "r")) == NULL) {
		fprintf(stderr, "Cannot open file...\n");
		exit(EXIT_FAILURE);
	}

	fseek(f, 0, SEEK_END);
	fileSize = ftell(f);
	fseek(f, 0, SEEK_SET);

	pString = (char *) malloc(fileSize * sizeof(char) + 1);
	if (pString == NULL) {
		fclose(f);
		fprintf(stderr, "Cannot allocate memory...\n");
		exit(EXIT_FAILURE);
	}
	if (fread(pString, sizeof(char), fileSize, f) <= 0) {
		free(pString);
		fclose(f);
		fprintf(stderr, "cannot read from file...\n");
		exit(EXIT_FAILURE);
	}
	pString[fileSize] = '\0';
	if (regcomp(&reg, argv[2], REG_EXTENDED) != 0) {
		free(pString);
		fclose(f);
		fprintf(stderr, "cannot compile regex...\n");
		exit(EXIT_FAILURE);
	}

	if (regexec(&reg, pString, 10, pMatch, 0) != 0) {
		regfree(&reg);
		free(pString);
		fclose(f);
```

```
            fprintf(stderr, "cannot match regex...\n");
            exit(EXIT_FAILURE);
      }

      for (i = 0; pMatch[i].rm_eo != -1; ++i) {
            for (j = pMatch[i].rm_so; j < pMatch[i].rm_eo; ++j)
                  printf("%c", pString[j]);

            putchar('\n');
      }

      regfree(&reg);
      free(pString);
      fclose(f);

      return 0;
}
```

6.2.2. scanner.h file

```
#ifndef _SCANNER_H_
#define _SCANNER_H_

#include <fstream>
#include <sstream>
#include <exception>

//ScannerException class
class ScannerException : public exception {
public:
      ScannerException(const char *pText) : m_text(pText)
      {}
      virtual const char *what()
      {
            return m_text.c_str();
      }
private:
      std::string m_text;
};



//TextSource class

class TextSource {
public:
      virtual int GetChar() = 0;
      virtual int ReadChar() = 0;
      virtual void PutBackChar() = 0;
};

class TextSourceFile : public TextSource {
public:
      TextSourceFile(const char *pFileName);
      virtual int GetChar()
      {
            int ch = m_f.rdbuf()->sgetc();
```

```
            return (ch == std::char_traits<char>::eof()) ? 0 : ch;
      }
      virtual int ReadChar()
      {
            int ch = m_f.rdbuf()->sbumpc();

            return (ch == std::char_traits<char>::eof()) ? 0 : ch;
      }

      virtual void PutBackChar()
      {
            m_f.rdbuf()->sungetc();
      }
private:
      std::ifstream m_f;
};

// TextSourceMemory class

class TextSourceMemory : public TextSource {
public:
      TextSourceMemory(const char *pString) : m_str(pString)
      {}
      virtual int GetChar()
      {
            int ch;

            ch = m_str.rdbuf()->sgetc();

            return (ch == -1) ? 0 : ch;
      }
      virtual int ReadChar()
      {
            int ch;

            ch = m_str.rdbuf()->sbumpc();

            return (ch == -1) ? 0 : ch;
      }

      virtual void PutBackChar()
      {
            m_str.rdbuf()->sungetc();
      }

private:
      std::istringstream m_str;
};

class Token {
public:
      virtual void GetToken(TextSource *pTextSource) = 0;
      virtual void Print() const = 0;

      enum TokenType {TT_Variable, TT_Constant,
TT_PunctuatorLeftParan, TT_PunctuatorRightParan,
            TT_PunctuatorSemicolon, TT_OperatorMinus,
TT_OperatorPlus, TT_OperatorMultiply,
            TT_OperatorDivision, TT_OperatorNot};
```

```
	TokenType GetTokenType() const
	{
		return m_tokenType;
	}
	const char *GetTokenString() const
	{
		return m_tokenString;
	}

protected:
	enum {MAX_TOKEN_STRING = 1024};

	TokenType m_tokenType;
	char m_tokenString[MAX_TOKEN_STRING];
};

//TokenVariable class

class TokenVariable : public Token {
public:
	virtual void GetToken(TextSource *pTextSource);
	virtual void Print() const;
};

//TokenConstant class

class TokenConstant : public Token {
public:
	virtual void GetToken(TextSource *pTextSource);
	virtual void Print() const;
};

// TokenPunctuator class

class TokenPunctuator : public Token {
public:
	virtual void GetToken(TextSource *pTextSource);
	virtual void Print() const;
};

// TokenOperator class

class TokenOperator : public Token {
public:
	virtual void GetToken(TextSource *pTextSource);
	virtual void Print() const;
};

//Scanner class

class Scanner {
public:
	Scanner(TextSource *pTextSource);
	Token *GetNextToken();
```

```
	enum CharCodeMask {CM_TypeMask = 0x0F, CM_AlNumMask = 0x10 };
	enum CharCode {CC_EOF, CC_Alpha, CC_Digit, CC_WhiteSpace, 
CC_Punctuator,
					CC_Operator, CC_Error, 
CC_AlNum = 0x10};

	static char ms_charMap[256];
private:
	void skipWS();
	TextSource *m_pTextSource;

	TokenVariable m_tokenVariable;
	TokenConstant m_tokenConstant;
	TokenPunctuator m_tokenPunctuator;
	TokenOperator m_tokenOperator;
};

#endif
```

6.2.3. scanner.cpp file

```
#include <iostream>
#include <cstdlib>
#include "scanner.h"

using namespace std;

// Global Definitions

char Scanner::ms_charMap[256]; // Character Map array

// TextSourceFile class

TextSourceFile::TextSourceFile(const char *pFileName)
{
	m_f.open(pFileName);
	if (!m_f)
		throw ScannerException("File cannot open");
}

// TokenVariable class

void TokenVariable::GetToken(TextSource *pTextSource)
{
	int i = 0;

	while ((Scanner::ms_charMap[pTextSource->GetChar()] & 
Scanner::CM_AlNumMask) == Scanner::CC_AlNum)
		m_tokenString[i++] = pTextSource->ReadChar();
	m_tokenString[i] = '\0';

}

void TokenVariable::Print() const
{
	cout << "Variable: " << m_tokenString << endl;
}
```

```
// TokenConstant class

void TokenConstant::GetToken(TextSource *pTextSource)
{
      int i = 0;

      while ((Scanner::ms_charMap[pTextSource->GetChar()] &
Scanner::CM_TypeMask) == Scanner::CC_Digit)
            m_tokenString[i++] = pTextSource->ReadChar();
      m_tokenString[i] = '\0';

      m_tokenType = TT_Constant;
}

void TokenConstant::Print() const
{
      cout << "Constant: " << m_tokenString << endl;
}

// TokenPunctuator class

void TokenPunctuator::GetToken(TextSource *pTextSource)
{
      switch (pTextSource->GetChar()) {
            case '(':
                  m_tokenType = TT_PunctuatorLeftParan;
                  break;
            case ')':
                  m_tokenType = TT_PunctuatorRightParan;
                  break;
            case ';':
                  m_tokenType = TT_PunctuatorSemicolon;
                  break;
      }
      m_tokenString[0] = pTextSource->ReadChar();
      m_tokenString[1] = '\0';
}

void TokenPunctuator::Print() const
{
      cout << "Punctuator: " << m_tokenString << endl;
}

// TokenOperator class

void TokenOperator::GetToken(TextSource *pTextSource)
{
            switch (pTextSource->GetChar()) {
                  case '+':
                        m_tokenType = TT_OperatorPlus;
                        break;
                  case '-':
                        m_tokenType = TT_OperatorMinus;
                        break;
                  case '*':
                        m_tokenType = TT_OperatorMultiply;
                        break;
                  case '/':
                        m_tokenType = TT_OperatorDivision;
                        break;
```

```
                case '!':
                    m_tokenType = TT_OperatorNot;
                    break;
            }

            m_tokenString[0] = pTextSource->ReadChar();
            m_tokenString[1] = '\0';
}

void TokenOperator::Print() const
{
      cout << "Operator: " << m_tokenString << endl;
}

// Scanner class

Scanner::Scanner(TextSource *pTextSource) :
m_pTextSource(pTextSource)
{
      if (ms_charMap[0] != CC_Error) {
            for (int i = 0; i < 256; ++i)
                  ms_charMap[i] = CC_Error;

            for (int i = 'a'; i <= 'z'; ++i)
                  ms_charMap[i] = CC_Alpha | CC_AlNum;
            for (int i = 'A'; i <= 'Z'; ++i)
                  ms_charMap[i] = CC_Alpha | CC_AlNum;
            for (int i = '0'; i <= '9'; ++i)
                  ms_charMap[i] = CC_Digit | CC_AlNum;

            ms_charMap[' '] = ms_charMap['\t'] = ms_charMap['\n'] =
ms_charMap['\r'] = CC_WhiteSpace;

            ms_charMap['_'] = CC_Alpha | CC_AlNum;

            ms_charMap[';'] = ms_charMap['('] =  ms_charMap[')'] =
CC_Punctuator;

            ms_charMap['+'] = ms_charMap['-'] = ms_charMap['*'] =
            ms_charMap['/'] = ms_charMap['!'] = CC_Operator;

            ms_charMap[0] = CC_EOF;
      }
}

void Scanner::skipWS()
{
      int ch = m_pTextSource->GetChar();

      while ((ms_charMap[ch] & CM_TypeMask) == CC_WhiteSpace) {
            m_pTextSource->ReadChar();
            ch = m_pTextSource->GetChar();
      }
}

Token *Scanner::GetNextToken()
{
      Token *pToken;

      skipWS();
```

```
        int a = m_pTextSource->GetChar();
        switch (ms_charMap[m_pTextSource->GetChar()] & CM_TypeMask) {
                case CC_Alpha:
                        pToken = &m_tokenVariable;
                        break;
                case CC_Digit:
                        pToken = &m_tokenConstant;
                        break;
                case CC_Punctuator:
                        pToken = &m_tokenPunctuator;
                        break;
                case CC_Operator:
                        pToken = &m_tokenOperator;
                        break;
                case CC_EOF:
                        return 0;
        }

        pToken->GetToken(m_pTextSource);

        return pToken;
}
```

6.3 APPENDIX C

6.3.1. parser.h file

```
#ifndef _PARSER_H_
#define _PARSER_H_

#include <stack>
#include "scanner.h"

//Parser class

class Parser {
public:
	Parser(Scanner *pScanner) : m_pScanner(pScanner)
	{}
	double Calculate();

private:
	void getNextToken()
	{
		m_pToken = m_pScanner->GetNextToken();
		if (!m_pToken)
			return;
		m_tokenType = m_pToken->GetTokenType();
	}
	void doExpression();
	void doAdditiveExpression();
	void doFactorExpression();
	void doUnaryExpression();
	void doPrimaryExpression();

	Scanner *m_pScanner;
	std::stack<double> m_stack;
	Token *m_pToken;
	Token::TokenType m_tokenType;
};


#endif
```

6.3.2. parser.cpp file

```
#include <iostream>
#include <cstdio>
#include <fstream>
#include <cstdlib>
#include "parser.h"

using namespace std;
```

```
double Parser::Calculate()
{
      double result;

      getNextToken();
      doExpression();
      result = m_stack.top();
      m_stack.pop();

      return result;
}

void Parser::doExpression()
{
    doAdditiveExpression();
}

void Parser::doAdditiveExpression()
{
      doFactorExpression();

      while (m_tokenType == Token::TT_OperatorPlus || m_tokenType ==
Token::TT_OperatorMinus)  {
            Token::TokenType tokenType = m_tokenType;

            getNextToken();
            doFactorExpression();
            double val1 = m_stack.top();
            m_stack.pop();
            double val2 = m_stack.top();
            m_stack.pop();
            m_stack.push((tokenType == Token::TT_OperatorPlus) ?
val1 + val2 : val2 - val1);
      }
}

void Parser::doFactorExpression()
{
      doUnaryExpression();

      while (m_tokenType == Token::TT_OperatorMultiply ||
m_tokenType == Token::TT_OperatorDivision)  {
            Token::TokenType tokenType = m_tokenType;

            getNextToken();
            doUnaryExpression();
            double val1 = m_stack.top();
            m_stack.pop();
            double val2 = m_stack.top();
            m_stack.pop();
            m_stack.push((tokenType == Token::TT_OperatorMultiply) ?
val1 * val2 : val2 / val1);
      }
}

void Parser::doUnaryExpression()
{
      doPrimaryExpression();

      if (m_tokenType == Token::TT_OperatorNot)  {
            Token::TokenType tokenType = m_tokenType;
```

```
            getNextToken();
            doUnaryExpression();
            double val = m_stack.top();
            m_stack.pop();
            m_stack.push(!val);
      }
}

void Parser::doPrimaryExpression()
{
      double number;

      switch (m_tokenType) {
            case Token::TT_Constant:
                  number = atof(m_pToken->GetTokenString());
                  m_stack.push(number);
                  getNextToken();
                  break;
            case Token::TT_PunctuatorLeftParan:
                  getNextToken();
                  doExpression();
                  if (m_pToken->GetTokenType() !=
Token::TT_PunctuatorRightParan) {
                        fprintf(stderr, "Fatal error: Paranthesis
mismatch!..\n");
                        exit(EXIT_FAILURE);
                  }
                  getNextToken();
                  break;
      }
}
```

6.4 APPENDIX D

6.4.1. interpreter.l file

```
/* --------------------------------------------------------------------
      Flex Source Code of the Sample Interpreter(interpreter.l)
--------------------------------------------------------------------*/
PRINT_COMMAND                 yazdir
EXIT_COMMAND                  cikis
IF_STATEMENT                  eger
ELSE_STATEMENT                degilse
WHILE_STATEMENT               iken
NUMBER                              (([0-9]+)|[0-9]*\.[0-9]+)
MULTI_OPERATORS_INC           \+\+
MULTI_OPERATORS_DEC           --
SINGLE_OPERATORS              [\+\-\*\/\^\(\)!=><\?:]
VARIABLES                     [_a-zA-Z][_a-zA-Z0-9]*
PUNCTUATORS                   ,|;|\{|\}
WHITESPACE                    [ \t\n]

%{
      #include <stdio.h>
      #include <string.h>
      #include <stdlib.h>
      #include <math.h>
      #include "interpreter.h"
      #include "interpreter.tab.h"

      /* static Function Prototypes */

      static int processPrintCommandLex(void);
      static int processExitCommandLex(void);
      static int processIfStatementLex(void);
      static int processElseStatementLex(void);
      static int processWhileStatementLex(void);
      static int processNumberLex(void);
      static int processMultiOperatorsIncLex(void);
      static int processMultiOperatorsDecLex(void);
      static int processSingleOperatorLex(void);
      static int processVariableLex(void);
      static int processPunctuatorsLex(void);
%}

%%

{PRINT_COMMAND}         { return processPrintCommandLex(); }
{EXIT_COMMAND}          { return processExitCommandLex(); }
{IF_STATEMENT}          { return processIfStatementLex(); }
{ELSE_STATEMENT}        { return processElseStatementLex(); }
{WHILE_STATEMENT}       { return processWhileStatementLex(); }
{NUMBER}                { return processNumberLex(); }
{MULTI_OPERATORS_INC}   { return processMultiOperatorsIncLex(); }
{MULTI_OPERATORS_DEC}   { return processMultiOperatorsDecLex(); }
{SINGLE_OPERATORS}      { return processSingleOperatorLex(); }
{VARIABLES}             { return processVariableLex(); }
{PUNCTUATORS}           { return processPunctuatorsLex(); }
{WHITESPACE}            /* Skip whitespace */
```

```
.                             { printf("Fatal error!..\n"); exit(1);
}
%%

/* Function Definitions */

static int processPrintCommandLex(void)
{
      return TOKEN_PRINT_COMMAND;
}

static int processExitCommandLex(void)
{
      return TOKEN_EXIT_COMMAND;
}

static int processIfStatementLex(void)
{
      return TOKEN_IF_STATEMENT;
}

static int processElseStatementLex(void)
{
      return TOKEN_ELSE_STATEMENT;
}

static int processWhileStatementLex(void)
{
      return TOKEN_WHILE_STATEMENT;
}

static int processNumberLex(void)
{
      yylval.value = atof(yytext);

      return TOKEN_NUMBER;
}

static int processMultiOperatorsIncLex(void)
{
      return TOKEN_OPERATOR_INC;
}

static int processMultiOperatorsDecLex(void)
{
      return TOKEN_OPERATOR_DEC;
}

static int processSingleOperatorLex(void)
{
      return *yytext;
}

static int processVariableLex(void)
{
      yylval.name = (char *) malloc(yyleng + 1);
      strcpy(yylval.name, yytext);

      return TOKEN_VARIABLE;
}
```

```
static int processPunctuatorsLex(void)
{
      return *yytext;
}
```

6.4.2. interpreter.y file

```
/* ---------------------------------------------------------------------
            Bison Source Code of Sample Interpreter (interpreter.y)
---------------------------------------------------------------------*/

%{
            #include <stdio.h>
            #include <string.h>
            #include <stdlib.h>
            #include <math.h>
            #include <assert.h>
            #include "interpreter.h"
%}

%union {
      double value;
      char *name;
      NODE *pNode;
};

%token <name> TOKEN_PRINT_COMMAND
%token <name> TOKEN_EXIT_COMMAND
%token <name> TOKEN_IF_STATEMENT
%token <name> TOKEN_ELSE_STATEMENT
%token <name> TOKEN_WHILE_STATEMENT
%token <value> TOKEN_NUMBER
%token <name>  TOKEN_VARIABLE
%token <name> TOKEN_OPERATOR_INC
%token <name> TOKEN_OPERATOR_DEC
%type <pNode> statement statement_list null_statement
simple_statement compound_statement
%type <pNode> command_statement if_statement while_statement
print_command_statement exit_command_statement
%type <pNode> expression assignment_expression
conditional_expression relational_expression
%type <pNode> additive_expression factor_expression
%type <pNode> unary_expression primary_expression power_expression

%%

input:      /* empty */
            | input     statement         { execute($2);
/*freeNodes($2); */}
;

statement:
            null_statement                { $$ = $1; }
      |     simple_statement        { $$ = processStatement($1,
STM_SIMPLE); }
      |     compound_statement            { $$ = processStatement($1,
STM_COMPOUND); }
      |     command_statement       { $$ = $1; }
      |     if_statement                  { $$ = processStatement($1,
STM_IF); }
```

```
      |     while_statement               { $$ = processStatement($1,
STM_WHILE); }
;

null_statement:
      ';'                                               { $$ = NULL; }
;

compound_statement:
            '{' '}'                                     { $$ = NULL; }
      |     '{'   statement_list '}'          { $$ = $2;  }
;

statement_list:
            statement    statement_list { $$ =
processStatementList($1, $2); }
      |     statement                         { $$ =
processStatementList($1, 0);  }
;

simple_statement:
      expression  ';'                                         {
$$ = $1; }
;

command_statement:
            print_command_statement                     { $$ =
processPrintCommand($1); }
      |     exit_command_statement                      { $$ =
processExitCommand(); }
;

print_command_statement:
            TOKEN_PRINT_COMMAND      expression ';'         { $$ =
$2; }
;

exit_command_statement:
            TOKEN_EXIT_COMMAND      ';'                        { }
;

if_statement:
            TOKEN_IF_STATEMENT      '(' expression ')' statement  {
$$ = processIfNode($3, $5, 0); }
      |     TOKEN_IF_STATEMENT      '(' expression ')' statement
      TOKEN_ELSE_STATEMENT    statement {$$ = processIfNode($3, $5,
$7); }
;

while_statement:
      TOKEN_WHILE_STATEMENT   '(' expression ')' statement  { $$ =
processWhileNode($3, $5); }
;

expression:
            assignment_expression
      { $$ = $1; }
;

assignment_expression:
```

```
            TOKEN_VARIABLE '=' assignment_expression       { NODE
*pNode = processSymbol($1); free($1); $$ =
processBinaryOperator(OP_ASSIGN, pNode, $3); }
      |     conditional_expression
      { $$ = $1; }
;

conditional_expression:
            relational_expression '?' conditional_expression ':'
conditional_expression { $$ = processTernaryOperator(OP_CONDITIONAL,
$1, $3, $5); }
      |     relational_expression
      { $$ = $1; }
;

relational_expression:
            relational_expression '>' additive_expression  {$$ =
processBinaryOperator(OP_GREATER, $1, $3); }
      |     relational_expression '<' additive_expression  {$$ =
processBinaryOperator(OP_LESS, $1, $3); }
      |     additive_expression
            { $$ = $1; }
;

additive_expression:
            additive_expression '+' factor_expression      { $$ =
processBinaryOperator(OP_ADD, $1, $3); }
      |     additive_expression '-' factor_expression      { $$ =
processBinaryOperator(OP_SUB, $1, $3); }
      |     factor_expression
      { $$ = $1; }
;

factor_expression:
            factor_expression '*' power_expression         { $$ =
processBinaryOperator(OP_MUL, $1, $3); }
      |     factor_expression '/' power_expression         { $$ =
processBinaryOperator(OP_DIV, $1, $3); }
      |     power_expression                                     {
$$ = $1; }
;

power_expression:
            power_expression '^' unary_expression          { $$ =
processBinaryOperator(OP_POW, $1, $3); }
      |     unary_expression                                     {
$$ = $1; }

;

unary_expression:
            '-' unary_expression                                 {
$$ = processUnaryOperator(OP_NEGATE, $2); }
      |     '!' unary_expression                                 {
$$ = processUnaryOperator(OP_NOT, $2); }
      |     TOKEN_OPERATOR_INC TOKEN_VARIABLE              { NODE
*pNode = processSymbol($2); free($2); $$ =
processUnaryOperator(OP_INC_PREFIX, pNode);}
      |     TOKEN_OPERATOR_DEC TOKEN_VARIABLE              { NODE
*pNode = processSymbol($2); free($2); $$ =
processUnaryOperator(OP_DEC_PREFIX, pNode);}
```

```
      |     TOKEN_VARIABLE TOKEN_OPERATOR_INC              { NODE
*pNode = processSymbol($1); free($1); $$ =
processUnaryOperator(OP_INC_POSTFIX, pNode);}
      |     TOKEN_VARIABLE TOKEN_OPERATOR_DEC              { NODE
*pNode = processSymbol($1); free($1); $$ =
processUnaryOperator(OP_DEC_POSTFIX, pNode);}
      |     primary_expression
      {$$ = $1; }
;

primary_expression:
            '(' expression ')'
      { $$ = $2; }
      |     TOKEN_NUMBER
      { $$ = processNumber($1); }
      |     TOKEN_VARIABLE
      { $$ = processSymbol($1); free($1); }
;

%%

int yywrap(void)
{
      return 1;
}

void yyerror(const char *str)
{
      printf("Fatal error: %s\n", str);
}

static NODE *getNode(void)
{
      NODE *pNode = (NODE *) malloc(sizeof(NODE));

      if (pNode == NULL)  {
            fprintf(stderr, "Fatal error: Not enouh memory!..\n");
            exit(EXIT_FAILURE);
      }

      return pNode;
}

static NODE *processPrintCommand(NODE *pNode)
{
      NODE *pNewNode = processStatement(pNode, STM_COMMAND);

      pNewNode->type.nodeStatement.extraInfo = COMSTM_PRINT;

      return pNewNode;
}

static NODE *processExitCommand(void)
{
      NODE *pNewNode = processStatement(NULL, STM_COMMAND);

      pNewNode->type.nodeStatement.extraInfo = COMSTM_EXIT;

      return pNewNode;
}
```

```
static NODE *processSymbol(const char *name)
{
	NODE *pNode = getNode();

	pNode->nodeType = NT_VARIABLE;
	pNode->type.nodeVariable.index = GetSetSymbol(name);

	return pNode;
}

static NODE *processNumber(double value)
{
	NODE *pNode = getNode();

	pNode->nodeType = NT_CONSTANT;
	pNode->type.nodeConstant.value = value;

	return pNode;
}

static NODE *processStatement(NODE *pNode, int stmType)
{
	NODE *pNewNode = getNode();

	pNewNode->nodeType = NT_STATEMENT;
	pNewNode->type.nodeStatement.stmType = stmType;
	pNewNode->type.nodeStatement.pFirstNode = pNode;

	return pNewNode;
}

static NODE *processStatementList(NODE *pNode1, NODE *pNode2)
{
	pNode1->type.nodeStatement.pNextCompoundNode = pNode2;

	return pNode1;
}

static NODE *processIfNode(NODE *pNode1, NODE *pNode2, NODE *pNode3)
{
	NODE *pNewNode = getNode();

	pNewNode->nodeType = NT_STATEMENT_IF;
	pNewNode->type.nodeStatementIf.pNodeExpression = pNode1;
	pNewNode->type.nodeStatementIf.pNodeTrue = pNode2;
	pNewNode->type.nodeStatementIf.pNodeFalse = pNode3;

	return pNewNode;
}

static NODE *processWhileNode(NODE *pNode1, NODE *pNode2)
{
	NODE *pNewNode = getNode();

	pNewNode->nodeType = NT_STATEMENT_WHILE;
	pNewNode->type.nodeStatementIf.pNodeExpression = pNode1;
	pNewNode->type.nodeStatementIf.pNodeTrue = pNode2;

	return pNewNode;
}
```

```
static NODE *processUnaryOperator(int opType, NODE *pNode1)
{
	NODE *pNode = getNode();

	pNode->nodeType = NT_OPERATOR;
	pNode->type.nodeOperator.opType = opType;
	pNode->type.nodeOperator.operands[0] = pNode1;

	return pNode;
}

static NODE *processBinaryOperator(int opType, NODE *pNode1, NODE
*pNode2)
{
	NODE *pNode = getNode();

	pNode->nodeType = NT_OPERATOR;
	pNode->type.nodeOperator.opType = opType;
	pNode->type.nodeOperator.operands[0] = pNode1;
	pNode->type.nodeOperator.operands[1] = pNode2;

	return pNode;
}

static NODE *processTernaryOperator(int opType, NODE *pNode1, NODE
*pNode2, NODE *pNode3)
{
	NODE *pNode = getNode();

	pNode->nodeType = NT_OPERATOR;
	pNode->type.nodeOperator.opType = opType;
	pNode->type.nodeOperator.operands[0] = pNode1;
	pNode->type.nodeOperator.operands[1] = pNode2;
	pNode->type.nodeOperator.operands[2] = pNode3;

	return pNode;
}

static double execute(NODE *pNode)
{
	if (pNode == NULL)
		return 0;

	switch (pNode->nodeType) {
		case NT_CONSTANT:
			return pNode->type.nodeConstant.value;

		case NT_VARIABLE:
			return GetSymbolByIndex(pNode-
>type.nodeVariable.index);
		case NT_OPERATOR:
			switch (pNode->type.nodeOperator.opType) {
				case OP_ADD:
					return execute(pNode-
>type.nodeOperator.operands[0]) + execute(pNode-
>type.nodeOperator.operands[1]);
				case OP_SUB:
					return execute(pNode-
>type.nodeOperator.operands[0]) - execute(pNode-
>type.nodeOperator.operands[1]);
				case OP_MUL:
```

```
                              return execute(pNode-
>type.nodeOperator.operands[0]) * execute(pNode-
>type.nodeOperator.operands[1]);
                        case OP_DIV:
                              return execute(pNode-
>type.nodeOperator.operands[0]) / execute(pNode-
>type.nodeOperator.operands[1]);
                        case OP_POW:
                              return pow (execute(pNode-
>type.nodeOperator.operands[0]), execute(pNode-
>type.nodeOperator.operands[1]));
                        case OP_NEGATE:
                              return -execute(pNode-
>type.nodeOperator.operands[0]);
                        case OP_NOT:
                              return !execute(pNode-
>type.nodeOperator.operands[0]);
                        case OP_INC_PREFIX:
                              {
                                    NODE *pIncNode;
                                    double result;

                                    pIncNode = pNode-
>type.nodeOperator.operands[0];
                                    result =
GetSymbolByIndex(pIncNode->type.nodeVariable.index);
                                    ++result;
                                    SetSymbolByIndex(pIncNode-
>type.nodeVariable.index, result);
                                    return result;
                              }
                        case OP_DEC_PREFIX:
                              {
                                    NODE *pIncNode;
                                    double result;

                                    pIncNode = pNode-
>type.nodeOperator.operands[0];
                                    result =
GetSymbolByIndex(pIncNode->type.nodeVariable.index);
                                    --result;
                                    SetSymbolByIndex(pIncNode-
>type.nodeVariable.index, result);
                                    return result;
                              }
                        case OP_INC_POSTFIX:
                              {
                                    NODE *pIncNode;
                                    double result;

                                    pIncNode = pNode-
>type.nodeOperator.operands[0];
                                    result =
GetSymbolByIndex(pIncNode->type.nodeVariable.index);
                                    SetSymbolByIndex(pIncNode-
>type.nodeVariable.index, result + 1);
                                    return result;
                              }
                        case OP_DEC_POSTFIX:
                              {
                                    NODE *pIncNode;
```

```
                                    double result;

                                    pIncNode = pNode-
>type.nodeOperator.operands[0];
                                    result =
GetSymbolByIndex(pIncNode->type.nodeVariable.index);
                                    SetSymbolByIndex(pIncNode-
>type.nodeVariable.index, result - 1);
                                    return result;
                              }
                        case OP_GREATER:
                              return execute(pNode-
>type.nodeOperator.operands[0]) > execute(pNode-
>type.nodeOperator.operands[1]);
                        case OP_LESS:
                              return execute(pNode-
>type.nodeOperator.operands[0]) < execute(pNode-
>type.nodeOperator.operands[1]);
                        case OP_ASSIGN:
                              {
                                    NODE *pLeftNode = pNode-
>type.nodeOperator.operands[0];
                                    NODE *pRightNode = pNode-
>type.nodeOperator.operands[1];
                                    double result;

                                    result = execute(pRightNode);

                                    SetSymbolByIndex(pLeftNode-
>type.nodeVariable.index, result);

                                    return result;
                              }
                  }
                  break;
            case NT_STATEMENT:
                  switch (pNode->type.nodeStatement.stmType) {
                        case STM_SIMPLE:
                              return execute(pNode-
>type.nodeStatement.pFirstNode);
                        case STM_COMMAND:
                              if (pNode-
>type.nodeStatement.extraInfo == COMSTM_PRINT) {
                                    printf("%f\n", execute(pNode-
>type.nodeStatement.pFirstNode));
                                    return 0;
                              }
                              else if (pNode-
>type.nodeStatement.extraInfo == COMSTM_EXIT)
                                    exit(0);
                              break;
                        case STM_IF:
                              {
                                    NODE *pNodeIf = pNode-
>type.nodeStatement.pFirstNode;

                                    if (execute(pNodeIf-
>type.nodeStatementIf.pNodeExpression))
                                          execute(pNodeIf-
>type.nodeStatementIf.pNodeTrue);
                                    else
```

```
                                          execute(pNodeIf-
>type.nodeStatementIf.pNodeFalse);

                                    return 0;
                              }
                        case STM_WHILE:
                              {
                                    NODE *pNodeWhile = pNode-
>type.nodeStatement.pFirstNode;

                                    while (execute(pNodeWhile-
>type.nodeStatementWhile.pNodeExpression))
                                          execute(pNodeWhile-
>type.nodeStatementWhile.pNodeTrue);
                                    return 0;
                              }
                        case STM_COMPOUND:
                              {
                              NODE *pNodeComp;

                                    for (pNodeComp = pNode-
>type.nodeStatement.pFirstNode; pNodeComp != NULL;

      pNodeComp = pNodeComp->type.nodeStatement.pNextCompoundNode)
                                          execute(pNodeComp);

                                    return 0;
                              }

                  }
      }

      return 0;
}

static void freeNodes(NODE *pNode)
{
      int i;

      if (pNode == NULL)
            return;

      /* ... */
}
```

6.4.3. interpreter.h file

```
#ifndef _INTERPRETER_H_
#define _INTERPRETER_H_

enum {NT_CONSTANT, NT_VARIABLE, NT_OPERATOR, NT_STATEMENT,
NT_STATEMENT_IF,
                        NT_STATEMENT_WHILE};

enum {OP_NEGATE, OP_NOT, OP_INC_PREFIX, OP_DEC_PREFIX,
OP_INC_POSTFIX, OP_DEC_POSTFIX,
      OP_POW, OP_MUL, OP_DIV, OP_ADD, OP_SUB, OP_GREATER, OP_LESS,
OP_CONDITIONAL, OP_ASSIGN
};
```

```
enum {STM_NULL, STM_SIMPLE, STM_COMPOUND, STM_COMMAND, STM_IF,
STM_WHILE };

enum {COMSTM_PRINT, COMSTM_EXIT };

/*Structure Declarations*/

typedef struct tagNODE_CONSTANT {
      double value;
} NODE_CONSTANT;

typedef struct tagNODE_VARIABLE {
      int index;
} NODE_VARIABLE;

typedef struct tagNODE_OPERATOR {
      int opType;
      struct tagNODE *operands[3];
} NODE_OPERATOR;

typedef struct tagNODE_STATEMENT {
      int stmType;
      int extraInfo;
      struct tagNODE *pFirstNode;
      struct tagNODE *pNextCompoundNode;
} NODE_STATEMENT;

typedef struct tagNODE_STATEMENT_IF {
      struct tagNODE *pNodeExpression;
      struct tagNODE *pNodeTrue;
      struct tagNODE *pNodeFalse;
} NODE_STATEMENT_IF;

typedef struct tagNODE_STATEMENT_WHILE {
      struct tagNODE *pNodeExpression;
      struct tagNODE *pNodeTrue;
} NODE_STATEMENT_WHILE;

typedef struct tagNODE {
      int nodeType;
      union {
            NODE_CONSTANT nodeConstant;
            NODE_VARIABLE nodeVariable;
            NODE_OPERATOR nodeOperator;
            NODE_STATEMENT nodeStatement;
            NODE_STATEMENT_IF nodeStatementIf;
            NODE_STATEMENT_WHILE nodeStatementWhile;
      } type;
} NODE;

/* Function Prototypes */

int yylex(void);
void yyerror(const char *str);
static NODE *getNode(void);
static NODE *processPrintCommand(NODE *pNode);
static NODE *processExitCommand(void);
static NODE *processSymbol(const char *str);
static NODE *processNumber(double value);
```

```
static NODE *processUnaryFunction(const char *fname, NODE
*pNodeArg);
static NODE *processBinaryFunction(const char *fname, NODE
*pNodeArg1, NODE *pNodeArg2);

static NODE *processStatement(NODE *pNode, int stmType);
static NODE *processStatementList(NODE *pNode1, NODE *pNode2);
static NODE *processIfNode(NODE *pNode1, NODE *pNode2, NODE
*pNode3);
static NODE *processWhileNode(NODE *pNode1, NODE *pNode2);
static NODE *processUnaryOperator(int opType, NODE *pNode1);
static NODE *processBinaryOperator(int opType, NODE *pNode1, NODE
*pNode2);
static NODE *processTernaryOperator(int opType, NODE *pNode1, NODE
*pNode2, NODE *pNode3);
static double execute(NODE *pNode);
static void freeNodes(NODE *pNode);

int GetSetSymbol(const char *name);
double SetSymbolByIndex(int index, double val);
double GetSymbolByName(const char *name);
double GetSymbolByIndex(int index);

#endif
```

6.4.4. interpreter.cpp file

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <algorithm>
#include <list>
#include <cstdlib>
#include <cassert>
#include "interpreter.hpp"

using namespace std;

/*--------------------------------------------
Global Data Definitions
--------------------------------------------*/

list<Symbol> g_variableList;

/*--------------------------------------------
Function Definitions
--------------------------------------------*/

extern "C" int GetSetSymbol(const char *name)
{
      list<Symbol>::iterator iter;
      int index;
      Symbol symbol(name);

      iter = find(g_variableList.begin(), g_variableList.end(),
symbol);
      if (iter == g_variableList.end()) {
```

```
            g_variableList.push_back(symbol);
            index = g_variableList.size() - 1;
      }
      else
            index = distance(g_variableList.begin(), iter);

      return index;
}

extern "C" double SetSymbolByIndex(int index, double val)
{
      list<Symbol>::iterator iter = g_variableList.begin();

      advance(iter, index);
      iter->m_val = val;

      return val;
}

extern "C" double GetSymbolByName(const char *name)
{
      list<Symbol>::iterator iter;
      Symbol symbol(name);

      iter = find(g_variableList.begin(), g_variableList.end(),
symbol);
      assert(iter != g_variableList.end());

      return iter->m_val;
}

extern "C" double GetSymbolByIndex(int index)
{
      list<Symbol>::iterator iter = g_variableList.begin();
      advance(iter, index);

      return iter->m_val;
}

int main(void)
{
      yyparse();

      return 0;
}
```

6.4.5. interpreter.hpp file

```
#ifndef _INTERPRETER_HPP_
#define _INTERPRETER_HPP_

#include <string>

struct Symbol {
      Symbol()
      {}
      Symbol(const char *name, double val = 0) : m_name(name),
m_val(val)
```

```
        {}
        bool operator ==(const Symbol &r) const
        {
                return r.m_name == m_name;
        }
        std::string m_name;
        double m_val;
};

/* Function Prototypes */

extern "C" int yyparse(void);
extern "C" int SetSymbol(const char *name);
extern "C" double SetSymbolByIndex(int index, double val);
extern "C" double GetSymbolByName(const char *name);
extern "C" double GetSymbolByIndex(int index);

#endif
```