CSC 488: Building a Compiler

For Beginners

Michael Liut
liutm@mcmaster.ca
michael.liut@utoronto.ca

January 24, 2017

Contents

1 Introduction

This document has been composed as supplemental material to assist the students of *CSC488H: Compilers and Interpreters* from The University of Toronto with their compiler construction project.

1.1 Software Design

If you recall any of your previous software design and specification courses, you will remember that above all else your primary goal is to ensure that your software works. Once you have a functioning program, you must strive for elegance, beauty and finesse. Remember, perfection is in the details! Finally, your concern must be efficiency. Optimize you program, but ensure that you never compromise the primary goal; your software must work!

1.2 Biting Off More Than You Can Chew

Over the years I have seen many students take on more than they can handle. Remember, many of you are taking 4-6 courses (some even 7 or 8) and building a compiler is a huge task that is not to be taken lightly.
One example I refer to often is the *inverted trapezoid model.* The layers (from top to bottom) are as follows:

1. Software Requirements Specification.
2. High-Level Architecture Design
3. Detailed Design
4. Final Product

The idea behind this model is that at each layer you will shave something off from your initial software requirements specification. As you move into a more detailed design you soon realize the importance of simplicity and will scale down your overall construction. Once you reach your final product (i.e. the compiler) you want to remember what is said in 1.1! If have already forgotten, go back and read it again!

Remember: you must ensure that your input language meets the minimum requires as per the assignment! You cannot "shave" these off.

1.3 What Does a Compiler Do? What Do I Need To Know?

In general, a compiler transforms an input language into a target language. For the purpose of this document I will not be discussing the three major components of a compiler in detail (the Front End, the Optimizer, and the Back End). Instead, I will be providing a toned down overview to allow you to hone in on key subsections for you to directly apply to your project. I will also imply some suggestions below; I strongly recommend you follow these suggestions for your first construction.

Firstly, a compiler is responsible for recognizing the syntax of a language. This means that the sentence(s) of the source program must be checked for validity. What you need to know is that parsing is where the source code, in essence, is converted into an Abstract Syntax Tree (hereinafter known as "AST") and semantically validated with your grammar.
A grammar is a set of rules that explicitly states how to properly form strings in a language's alphabet and ensure they are valid with respect to the language's syntax. What you need to know is that you will be better off conducting a top-down parsing approach and creating a predictive grammar (thereby being $LL(1)$). Alternatively, you may opt for $LALR(1)$, a superset of LL(*), but I think a predictive

grammar will reduce your work in later sections (a.k.a. sprints). Also, use of a predictive grammar will remove the risk of memory leaks.

At this point you may be wondering how to transform your code from a high-level (source) language into your desired lower-level (target) language. You may also be wondering how to implement some high-level optimization. What you need to know is that your AST has to be transformed to represent a more efficient computation utilizing the same semantics. Eliminating superfluous local assignments, early calculations of constant expressions and common subexpressions are just a few examples of trivially redundant high-level code that can be optimized at an early stage. **This is an extremely important part of a compiler and this brief synopsis only scratches the surface!**

Now you may be asking yourselves, how do I generate the low-level (target) language? The general idea is to utilize the parse tree by transforming it into a low-level language; commonly x86, MIPS, or ARM. This concept is widely used but completely dependent on the source language and intended target language. With respect to optimization at a low-level stage (also commonly referred to as peephole optimization), what you need to know is that, the low-level code is scanned for inefficiencies which are then modified and corrected.

Note: both code generation and low-level optimization can be repeated several times.

Let's look at an example:

> If we look at the GNU Compiler Collection (hereinafter known as "GCC"), GCC repeats the code generation and low-level optimization stages many times. It transforms the high-level C code into an intermediary (lower-level) language (which is platform independent). It further optimizes the intermediary code and then transforms this intermediary language into its target (and true low-level) language (e.g. x86, MIPS, ARM, etc...). Optimization occurs several more times, and finally, linking occurs (i.e. the resolution of references to other modules).

2 The DOs and DON'Ts of Compiler Construction

This section will provide you with a brief list of common tips to ease your compiler construction experience.

2.1 DO

1. **Define your language well!**
   Ensure that your language is defined well syntactically and semantically. If it is not, it can turn out to be more work on your part.
2. **Choose a good syntax analyzer!**
   There are many that you can choose from, we will show you how to use FLEX. It is fast, easy to use and open-source.
3. **Choose a good parser!**
   There are many that you can choose from, we will show you how to use CUP; a LALR parser generator. You can use CUP with your predictive grammar. Alternative options to CUP that I have worked with are: ANTLR3/4 (LL(*)), Bison (LALR(1)), and YACC (LALR(1)).
4. **Test your code!**
   Establish a set of test cases and be sure to run them regularly. Remember, you may use regular expressions for validity checking too.

5. **Correct errors before moving forward!**
   Running those test cases is one thing, but if they fail you need to fix what is broken prior to moving on. This can be slow, long and tedious; but it is essential!

2.2 DON'T

1. **Code before having your language and design specifications!**
   Skipping to the coding phase without verifying your language and design specifications can turn out to be detrimental.
2. **Get overwhelmed by thinking of the project as a whole from the beginning!**
   This project is broken up into sprints for a reason.
3. **Shoot down others ideas!**
   Give them a chance! Their ideas can prove to be useful!
4. **Make a syntax analyzer!**
   There are many that you can choose from, it is difficult to make one from scratch and quite frankly not the easiest thing to do!
5. **Make a parser!**
   There are many that you can choose from, it is difficult to make one from scratch and quite frankly not the easiest thing to do!

3 Using LEX/FLEX and CUP/YACC

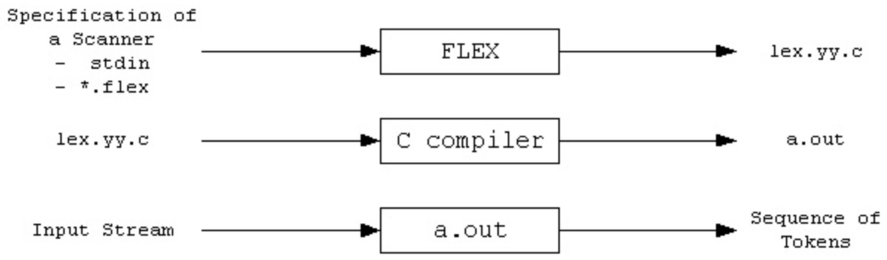
This section will be used to briefly explain a scanner and parser that you may use for your project.

3.1 LEX/FLEX [4]

The scanner performs lexical analysis of your program by reading in the input language as a sequence of characters and recognizing them as tokens. FLEX (Fast LEXical analyzer generator) is used to generate scanners. This is done by identifying the pattern in the text of a certain language. For example, a digit would represent $[0-9]$.

What you need to know: is that LEX and FLEX for the most part are the same. The difference is that LEX affords you the capability to use your own input code and modify the character stream, whereas FLEX does not allow you to do so. You can use either or.

In general, FLEX is utilized in the following manner:



FLEX begins by reading a specification of a scanner (either an input file **.lex* or from standard input) which it uses to generate a C target file as output *lex.yy.c*. This outputted C file is then compiled and linked using the "-lfl" library, which produces an executable file *a.out*. This executable file analyzes its input stream and transforms it into a sequence of tokens.

What you need to know:

- **.lex* uses regular expressions and C code.
- *lex.yy.c* defines a routine *yylex()* that uses the specification to recognize tokens.
- *a.out* is the scanner.

3.1.1 Example

Give a lex/flex input file to generate a scanner that recognizes the following tokens:

1. Java identifier
2. C unsigned integer
3. C literal string

and that 'eats' multi-line comments, where the comment starts with "[|[" and ends with "]|]". The nested multi-line comments are not allowed (similarly as in C/C++/Java). The scanner must treat the comment as white space.

3.1.2 Solution

Given the requirements in 3.1.1., you should have a solution that looks like this:

Example Solution

```
%{
        /* Michael Liut */
        /* UofT - CSC488  */
        /* COMMANDS TO EXECUTE:
         *     lex example.lex
         *     gcc -ll lex.yy.c
         *     ./a.out < test
        */
%}

%option noyywrap
%x checkComments

%{
        /* REGEX Variable Initialization */
%}
ws                                 [ \n]|[\n]|[\r\n]|[\t]|[\r]
begComment                 "[|["
endComment                 "]|]"
integers                 [0-9]+
cLiteral                 \"(\\.|[^"])*\"
jIdentifiers         (([a-zA-Z_$])([a-zA-Z0-9_$])*)
```

```
%%

%{
        /* Begin Function */
%}
{begComment} {BEGIN(checkComments);}

%{
        /* Checks for comment end without a start -- Returns -1 */
%}
{endComment} {fprintf(stderr, "Error: Found comment end without a beginning!\n"); return -1;}

%{
        /* Error Checking -- Check for opening comment inside of a comment */
%}
<checkComments>{endComment} {BEGIN(INITIAL);}
<checkComments>{begComment} {fprintf(stderr, "Error: Found [|[ inside another
                                                         [|[ comment!\n"); return -1;}

%{
        /* Removes the comments & comment content */
%}
<checkComments>(.|\n)
<checkComments><<EOF>> {fprintf(stderr, "Error: Found [|[ without closing ]|]!\n"); return -1;}

{integers} {printf("An integer (%s): %d \n", yytext, atoi(yytext));}
{jIdentifiers} {printf("Java identifier: %s \n", yytext);}
{cLiteral} {printf("C Literal: %s \n", yytext);}
{ws} {}
. {printf("ERROR DETECTED %s \n", yytext);}

%%
int main(int argc, char* argv[]){
    yylex();
}
```

Some details with respect to the segment of code above:

1. "noyywrap" is used when there is only one input file, you likely will not turn on this feature. However, as Flex declares the function *yywrap* we must define it.
2. "$\%x$" indicates that only prefixed rules with the state's name will be active when the scanner is in that state. This is an exclusive property.

So you may be looking at this solution and trying to understand what is being accomplished? This file is your flex file, you are using it to direct your scanner into what it needs to check and/or remove from the source file (i.e. your input file). The flex file is used to construct your scanner *a.out*. This flex file is eliminating multi-line comments as they do not need to be parsed.

What you need to know is that the a.out file generated from this process is your scanner.

3.2 CUP [1, 3]

3.2.1 Recursive Patterns and Context Free Grammars [1]

"A context-free grammar is a set of recursive rewriting rules (or productions) used to generate patterns of strings. Context-free grammars are often used to define the syntax of programming languages.

A parse tree displays the structure used by a grammar to generate an input string. Parse trees are typically used within a compiler to describe the structure of an input program in terms of the syntactic rules used to define valid programs.

A parser is an algorithm that determines whether a given input string is in a language (and, as a side-effect, usually produces a parse tree for the input). There is a mechanical procedure for generating a parser from a given context-free grammar."

A Context Free Grammar consists of the following components:

1. a set of terminal symbols, which are the characters of the alphabet that appear in the strings generated by the grammar.
2. a set of non-terminal symbols, which are placeholders for patterns of terminal symbols that can be generated by the non-terminal symbols.
3. a set of productions, which are rules for replacing (or rewriting) non-terminal symbols (on the left side of the production) in a string with other non-terminal or terminal symbols (on the right side of the production).
4. a start symbol, which is a special non-terminal symbol that appears in the initial string generated by the grammar. By convention the start symbol is usually the left-hand side of the first production.

To generate a string of terminal symbols from a Context Free Grammar, we:

1. begin with a string consisting of the start symbol;
2. apply one of the productions with the start symbol on the left-hand side, replacing the start symbol with the right-hand side of the production;
3. repeat the process of selecting non-terminal symbols in the string, and replacing them with the right-hand side of some corresponding production, until all non-terminals have been replaced by terminal symbols. The resulting sequence of strings is called a *derivation*.

More details and examples on the information extracted from Dr. C. Brown's document can be found here.

3.2.2 Ambiguous Grammars [1]

An Ambiguous Grammar is one which there are two different parse trees for the same terminal string. For example, let's say a grammar for balanced parentheses looks like this:

$$P \rightarrow (\; P \;) \mid P \; P \mid \epsilon$$

where "|" is the notational shorthand for "or" and "ϵ" denotes an empty string (i.e. an empty right-hand side).

What you need to know is that you can prove any grammar is ambiguous by demonstrating two parse trees for the same terminal string.

In this particular example, we can also prove that the grammar is ambiguous, let's look for the empty string "ϵ".

Case 1. $P \rightarrow \epsilon$

Case 2. $P \rightarrow P\ P \rightarrow \epsilon\ \epsilon$

Now say you are still not convinced that this grammar is ambiguous, maybe you believe that you will fair better with "( )". Lets try.

Case 1. $P \rightarrow P\ P \rightarrow (\ P\ )\ \epsilon \rightarrow (\ \epsilon\ )\ \epsilon$

Case 2. $P \rightarrow P\ P \rightarrow \epsilon\ (\ P\ ) \rightarrow \epsilon\ (\ \epsilon\ )$

What would the unambiguous grammar look like?

$$P \rightarrow (\ P\ )\ P \mid \epsilon$$

3.2.3 The Problem with Ambiguous Grammars [1]

"A parse tree is supposed to display the structure used by a grammar to generate an input string. This structure is not unique if the grammar is ambiguous. A problem arises if we attempt to impart meaning to an input string using a parse tree; if the parse tree is not unique, then the string has multiple meanings.

We typically use a grammar to define the syntax of a programming language. The structure of the parse tree produced by the grammar imparts some meaning on the strings of the language."

What you need to know is that "if the grammar is ambiguous, the compiler has no way to determine which of two meanings to use. Thus, the code produced by the compiler is not fully determined by the program input to the compiler."

More details and examples on the information extracted from Dr. C. Brown's document can be found here.

3.2.4 Grammar Example [3]

Let's say that we are trying to define string expressions for a new programming language.
The terminals are as follows:

1. **STRLIT** is a token ***string literal***.
2. **SID** is a token ***string identifier*** representing a name of a string variable or a name of a method returning a string value.

3. '[' is a token.
4. ']' is a token.
5. ':' is a token.
6. '(' is a token.
7. ')' is a token.
8. ',' is a token.
9. '+' is a token.

For simplicity's sake, we will treat the non-terminal ***iexpr*** as a terminal (i.e. we will not provide a definition for ***iexpr*** – which is representing an integer expression).

Given the following notation of string expressions below, give a Context-Free Unambiguous Grammar in Backus-Naur Form.

1. **STRLIT** is a string expression (meaning a string literal).
2. **SID** is a string expression (meaning a name of a string variable).
3. **SID()** is a string expression (meaning a call to a method returning a string value).
4. if X, X_1, ..., X_n are string expressions, then so are the following sentential forms:
   (a) **SID(** X_1 **)** represents a call to a method with one string argument returning a string value.
   (b) **SID(** X_1, X_2 **)** represents a call to a method with two string arguments returning a string value.
   (c) **SID(** X_1, X_2, X_3 **)** represents a call to a method with three string arguments returning a string value.
   (d) **SID(** X_1, ..., X_n **)** represents a call to a method with n string arguments returning a string value.
   (e) X**[iexpr]** represents the symbol of the value of X at position **iexpr**.
   (f) X**[iexpr$_1$:iexpr$_2$]** represents the symbol of the value of X at position **iexpr$_1$** to position **iexpr$_2$**.
   (g) X**[:iexpr]** represents the prefix of the value of X from position 0 to position **iexpr**.
   (h) X**[iexpr:]** represents the suffix of the value of X from position **iexpr** to the end position.
   (i) $X_1 + X_2$ represents the concatenation of the value of X_1 with the value of X_2.

3.2.5 Grammar Solution

Context-Free Grammar in Backus-Naur Form:

$\langle strfun \rangle$ ::= SID $\langle sid\_tail \rangle$

$\langle sid\_tail \rangle$ ::= ϵ
| '(' $\langle args \rangle$

$\langle args \rangle$ ::= ‘)’
  | $\langle strexp \rangle$ $\langle args\_tail \rangle$

$\langle args\_tail \rangle$ ::= ‘)’
  | ‘,’ $\langle strexp \rangle$ $\langle arg\_tail \rangle$

$\langle strexp \rangle$ ::= STRLIT $\langle strlit\_tail \rangle$
  | SID $\langle sid\_tail\_one \rangle$

$\langle strlit\_tail \rangle$ ::= ϵ
  | ‘[’ $\langle rng \rangle$ $\langle strlit\_tail \rangle$
  | ‘+’ $\langle strexp \rangle$

$\langle sid\_tail\_one \rangle$ ::= ϵ
  | ‘(’ $\langle sid\_tail\_two \rangle$

$\langle sid\_tail\_two \rangle$ ::= ‘)’ $\langle sid\_tail\_three \rangle$
  | $\langle strexp \rangle$ $\langle sid\_tail\_four \rangle$

$\langle sid\_tail\_three \rangle$ ::= ϵ
  | ‘[’ $\langle rng \rangle$ ‘]’ $\langle strlit\_tail\_three \rangle$

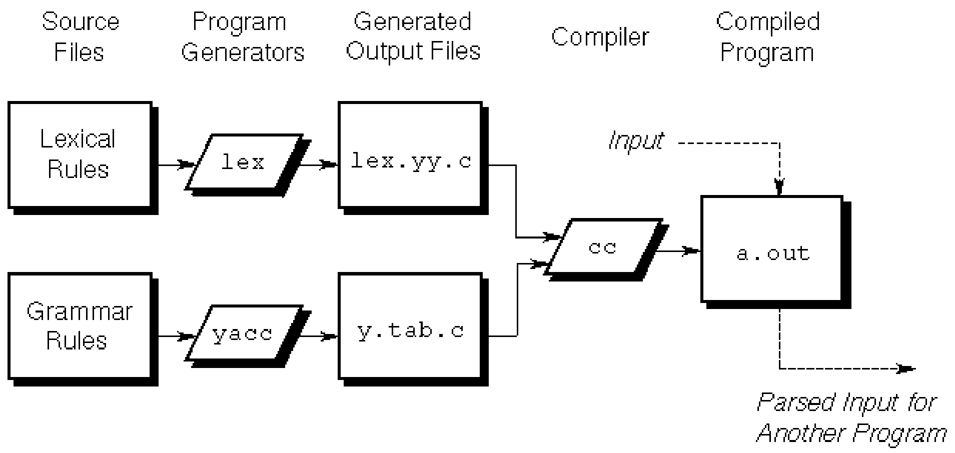
$\langle sid\_tail\_four \rangle$ ::= ‘)’ $\langle sid\_tail\_three \rangle$
  | ‘,’ $\langle sid\_tail\_four \rangle$

$\langle rng \rangle$ ::= ‘:’ IEXPR ‘]’
  | IEXPR ‘:’ $\langle rng\_tail \rangle$

$\langle rng\_tail \rangle$ ::= ‘]’
  | ‘:’ IEXPR

3.2.6 CUP/YACC [2, 5]

Yet **A**nother **C**ompiler **C**ompiler (hereinafter referred to as YACC), is a **L**ook **A**head **L**eft-to-**R**ight (hereinafter referred to as LALR) parser generator for C++ and C#. In general, FLEX and YACC are utilized in the following manner:



Construction of **U**seful **P**arsers (hereinafter referred to as CUP) is a LALR parser generator for Java. YACC is intended to be CUP's predecessor. The Technische Universität München is currently responsible for maintaining CUP. More details and examples can be found here.

3.2.7 Grammar Solution in YACC

Using YACC, verify the grammar (i.e. it should not give you any shift/reduce error).

Example Solution

```
/*
        * Michael Liut
        * UofT - CSC488
        * COMMANDS TO EXECUTE:
        *        yacc -d example.yacc
        *       gcc lex.yy.c y.tab.c -ll
        *       ./a.out < test
*/

%{
        #include <stdio.h>
        extern int yylex(void);
        void yyerror(char const *s) { fprintf(stderr, "%s\n", s); }
%}

%token STRLIT SID IEXPR
%%

/* STRING FUNCTION */
strfun:        SID sid_tail;

sid_tail: /* empty */ | '(' args;

/* Arguments */
args: ')' | strexp args_tail;

args_tail: ')' | ',' strexp args_tail;

/* STRING EXPRESSION */
strexp:        STRLIT strlit_tail | SID sid_tail_one;

strlit_tail: /* empty */ | '[' rng strlit_tail | '+' strexp;

sid_tail_one: /* empty */ | '(' sid_tail_two;

sid_tail_two: ')' sid_tail_three | strexp sid_tail_four;

sid_tail_three: /* empty */ | '[' rng ']' sid_tail_three | '+' strexp;

sid_tail_four: ')' sid_tail_three | ',' sid_tail_four;

/* RNG for Integer Expressions */

rng: ':' IEXPR ']' | IEXPR ':' rng_tail;

rng_tail: ']' | ':' IEXPR;
%%

int main() {
        yyparse();
}
```

3.2.8 Grammar Solution in C++

```
/*
        * Michael Liut
        * UofT - CSC488
        * Recursive Descent Parser for my LL(1) grammar
        * NOTE: Followed skeleton format located in Sebesta's
        *               Concepts of Programming Languages Textbook.
        *               Edition 10. Pages 172 - 177.
        * NOTE: Dr. Franek's "Help" code utilized here too.
 */

/*************************************************************************/
/********************************  BEGIN  ********************************/
/*************************************************************************/

/* Including Base Packages */
#include <stdio.h>
#include <ctype.h>

/* Global Variable Declarations */
int charClass;
char lexeme [100];
char nextChar;
int lexLen;
int token;
int nextToken;
FILE *in_fp, *fopen();

/* Blue Printing -- Function Declarations */
void addChar();
void getChar();
void getNonBlank();
int lex();

/* Character Classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99

/* Defining Token Codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26

/* Defining Multi-Purpose Functions */
#define isdigit(x) ((x)>='0'&&(x)<='9')
#define isvar(x) (((x)>='a'&&(x)<='z')||((x)>='A'&&(x)<='Z'))
```

```
#define isws(x) ((x)==' '||(x)=='\n'||(x)=='\t')
#define EatGarbage() while(isws(s[*spp])) (*spp)++
#define SavePosition() spp1 = *spp
#define RestorePosition() *spp = spp1

/*************************************************************************/
/*******************************  PARSER  ********************************/
/*************************************************************************/
/*int Parse(char* s) {
    int spp;

    spp = 0;
    if(!strFun(s, &spp))
        return 0;

    while(isws(s[spp])) spp++;

    if (s[spp] == '\0')
        return 1;
    else
        return 0;
} //end Parse Function*/

/*************************************************************************/
/*******************************  GRAMMAR  *******************************/
/*************************************************************************/
/*
 *      <strfun> ::= SID <sid_tail>
 *      <sid_tail> ::= Epsilon | "(" <args>
 *      <args> ::= ")" | <strexp> <args_tail>
 *      <args_tail> ::= ")" | "," <strexp> <arg_tail>
 *      <strexp> ::= STRLIT <strlit_tail> | SID <sid_tail_one>
 *      <strlit_tail> ::= Epsilon | "[" <rng> <strlit_tail> | "+" <strexp>
 *      <sid_tail_one> ::= Epsilon | "(" <sid_tail_two>
 *      <sid_tail_two>  ::= ")" <sid_tail_three>
 *      <sid_tail_three> ::= Epsilon | "[" <rng > "]" <strlit_tail_three>
 *      <sid_tail-four> ::= ")" <sid_tail_three> | "," <sid_tail_four>
 *      <rng> ::= ":" IEXPR "]" | IEXPR ":" <rng_tail>
 *      <rng_tail> ::= "]" | ":" IEXPR
 */
/*************************************************************************/
/*************************************************************************/

int sidTail(char* s, int *spp);

// <strexp> ::= STRLIT <strlit_tail> | SID <sid_tail_one>
int strExp(char* s, int *spp) ;

// Checks if it is an integer
int isIEXPR(char* s, int *spp) ;

// Checks if it is a Java Identifier
int isSID(char* s, int *spp) ;

// <strfun> ::= SID <sid_tail>
int strFun(char* s, int *spp) {
    int spp1, spp2;

    SavePosition();
    EatGarbage();

    if(!isSID(s, spp)) {
        RestorePosition();
        return 0;
    }
    spp2 = *spp;
    if(!sidTail(s, &spp2)) {
        RestorePosition();
        return 0;
    }
    return 1;
}

// Checks if it is a string literal
int isSTRLIT(char* s, int *spp) {
    int spp1;

    SavePosition();
    EatGarbage();

    if(s[*spp]=='\"') {
        (*spp)++;
        while(s[*spp]!='\"')
            (*spp)++;
        (*spp)++;
        return 1;
    } else {
        RestorePosition();
        return 0;
    }
}

// <args_tail> ::= ")" | "," <strexp> <arg_tail>
int argsTail(char* s, int *spp) {
    int spp1, spp2;

    SavePosition();
    EatGarbage();

    // ")"
    if(s[*spp] == ')') {
        return 0;
    }

    if(s[*spp] == '\0') {
        RestorePosition();
        return 0;
    }

    if(s[*spp] == ',') {
        // <strexp> <args_tail>
        if(!strExp(s, spp)) {
            RestorePosition();
            return 0;
        }

        spp2 = *spp;

        if(!argsTail(s, &spp2)) {
            RestorePosition();
            return 0;
        }
    }
    return 1;
```

```
}

// <args> ::= ")" | <strexp> <args_tail>
int args(char* s, int *spp) {
    int spp1, spp2;

    SavePosition();
    EatGarbage();

    // ")"
    if(s[*spp] == ')') {
        return 0;
    }

    if(s[*spp] == '\0') {
        RestorePosition();
        return 0;
    }

    // <strexp> <args_tail>
    if(!strExp(s, spp)) {
        RestorePosition();
        return 0;
    }

    spp2 = *spp;

    if(!argsTail(s, &spp2)) {
        RestorePosition();
        return 0;
    }
    return 1;
}

// <sid_tail_one> ::= Epsilon | "(" <sid_tail_two>
int sidTailOne(char* s, int *spp) {
    int spp1;

    SavePosition();
    EatGarbage();

    // "(" <sid_tail_two>
    if(s[*spp] == '(') {
        (*spp)++;
        if(!args(s, spp)){
            RestorePosition();
            return 0;
        }
        (*spp)--;
    }

    return 1; // Epsilon

}

// <strexp> ::= STRLIT <strlit_tail> | SID <sid_tail_one>
int strExp(char* s, int *spp) {
    int spp1, spp2;

    SavePosition();
    EatGarbage();
    // STRLIT <strlit_tail>
    if(!isSTRLIT(s, spp)) {
        return 0;
    }

    if(s[*spp] == '\0') {
        RestorePosition();
        return 0;
    }

    // SID <sid_tail_one>
    if(!strExp(s, spp)) {
        RestorePosition();
        return 0;
    }

    spp2 = *spp;

    if(!sidTailOne(s, &spp2)) {
        RestorePosition();
        return 0;
    }
    return 1;
}

// <sid_tail> ::= Epsilon | "(" <args>
int sidTail(char* s, int *spp) {
    int spp1;

    SavePosition();
    EatGarbage();

    // "(" <args>
    if(s[*spp] == '(') {
        (*spp)++;
        if(!args(s, spp)){
            RestorePosition();
            return 0;
        }
        (*spp)--;
    }
    return 1; // Epsilon
}

// <rng_tail> ::= "]" | ":" IEXPR
int rngTail(char* s, int *spp) {
    int spp1, spp2;

    SavePosition();
    EatGarbage();

    if(s[*spp]!=']') {
        RestorePosition();
        return 0;
    }

    // ":" IEXPR "]"
    if(s[*spp]==':') {
        (*spp)++;
        if(!isIEXPR(s, spp)) {
            (*spp)++;
            if (s[*spp]==']'){
                RestorePosition();
                return 0;
            }
        }
    }
    return 1;
}

// <rng> ::= ":" IEXPR "]" | IEXPR ":" <rng_tail>
int rng(char* s, int *spp) {
    int spp1, spp2;

    SavePosition();
    EatGarbage();

    // ":" IEXPR "]"
    if(s[*spp]==':') {
        (*spp)++;
        if(!isIEXPR(s, spp)) {
            (*spp)++;
            if (s[*spp]==']'){
                RestorePosition();
                return 0;
            }
        }
    }

    // IEXPR ":" <rng_tail>
    if(!isIEXPR(s, spp)) {
        if(s[*spp]==':') {
            (*spp)++;
            if(!rngTail(s, spp)) {
                RestorePosition();
                return 0;
            }
        }
    }
    return 1;
}

// <strlit_tail> ::= Epsilon | "[" <rng> <strlit_tail> | "+" <strexp>
int strlitTail(char* s, int *spp) {
    int spp1, spp2;

    SavePosition();
    EatGarbage();

    // "[" <rng> <strlit_tail>
    if(s[*spp]=='[') {
        (*spp)++;
        if(!rng(s, spp)) {
            RestorePosition();
            return 0;
        }
        if(!strlitTail(s, spp)){
            RestorePosition();
            return 0;
        }
    }

    // "+" <strexp>
    if(s[*spp]=='+') {
        (*spp)++;
        if(!strExp(s, spp)) {
            RestorePosition();
            return 0;
        }
    }

    return 1; // Epsilon

}

// <sid_tail_three> ::= Epsilon | "[" <rng> "]" <strlit_tail_three>
int sidTailThree(char* s, int *spp) {
    int spp1, spp2;

    SavePosition();
    EatGarbage();

    // "[" <rng> "]" <strlit_tail_three>
    if(s[*spp]=='[') {
        (*spp)++;
        if(!args(s, spp)) {
            RestorePosition();
            return 0;
        }

        if(s[*spp]!=']') {
            RestorePosition();
            return 0;
        }

        (*spp)++;

        if(!sidTailThree(s, spp)) {
            RestorePosition();
            return 0;
        }
    }

    return 1; // Epsilon

}

// <sid_tail_two>  ::= ")" <sid_tail_three>
int sidTailTwo(char* s, int *spp) {
    int spp1, spp2;

    SavePosition();
    EatGarbage();

    // ")" <sid_tail_three>
    if(s[*spp] == ')') {
        (*spp)++;
        if(!sidTailThree(s, spp)){
            RestorePosition();
            return 0;
        }
        (*spp)--;
    }
    return 1;
}
```

```
// <sid_tail-four> ::= ")" <sid_tail_three> | "," <sid_tail_four>
int sidTailFour(char* s, int *spp) {
    int spp1;

    SavePosition();
    EatGarbage();

    // ")" <sid_tail_three>
    if(s[*spp] == ')') {
        if(!sidTailFour(s, spp)) {
            RestorePosition();
            return 0;
        }
    }

    // "," <sid_tail_four>
    if(s[*spp] == ',') {
        if(!sidTailFour(s, spp)) {
            RestorePosition();
            return 0;
        }
    }
    return 1;
}

/***************************************************************************/
/***************************** JID/INT/STRLIT ******************************/
/****************************** CHECKER ************************************/
/***************************************************************************/

// Checks if it is an integer
int isIEXPR(char* s, int *spp) {
    int spp1, i;

    SavePosition();
    EatGarbage();

    if (s[*spp]=='0') {
        (*spp)++;
        return 1;
    }

    for(i = 0; i < 9 && isdigit(s[*spp]); i++, (*spp)++);

    if (i == 0) {
        RestorePosition();
        return 0;
    }

    return 1;
}

// Checks if it is a Java Identifier
int isSID(char* s, int *spp) {
    int spp1;

    SavePosition();
    EatGarbage();

    if((s[*spp]=='_')||(s[*spp]=='$')||isvar(s[*spp])) {
        (*spp)++;
        while(!(isws(s[*spp]))) {
            if((s[*spp]=='_')||(s[*spp]=='$')||isvar(s[*spp])||isdigit(s[*spp])){
                (*spp)++;
            } else {
                RestorePosition();
                return 0;
            }
        }
        return 1;
    } else {
        RestorePosition();
        return 0;
    }
}

/****************************************************************************/
/*****************************    MAIN  METHOD   *****************************/
/****************************************************************************/

// main() {
// /* Open the input data file and process its contents */
//       if ((in_fp = fopen("front.in", "r")) == NULL)
//               printf("ERROR - cannot open front.in \n");
//       else {
//               getChar();
//               do {
//                       lex();
//               } while (nextToken != EOF);
//       }
// } // end of main method

int main (int argc, char ** argv){
    ++argv;
    --argc;
    if (argc > 0){
        in_fp = fopen(argv[0], "r");

        if (in_fp == NULL)
            printf("ERROR - cannot open file! \n");
    } else {
        in_fp = stdin;
    }
    printf("Test Passed!\nHave a great day! :-) \n");
}

/****************************************************************************/
/***************************   OTHER FUNCTIONS   *****************************/
/****************************************************************************/

/* lookup Function
 *              "Looks up" operators and parentheses, the returns the token
 */
int lookup(char ch) {
    switch (ch) {
        case '(':
            addChar();
            nextToken = LEFT_PAREN;
            break;
        case ')':
            addChar();
            nextToken = RIGHT_PAREN;
            break;
        case '+':
            addChar();
            nextToken = ADD_OP;
            break;
        case '-':
            addChar();
            nextToken = SUB_OP;
            break;
        case '*':
            addChar();
            nextToken = MULT_OP;
            break;
        case '/':
            addChar();
            nextToken = DIV_OP;
            break;
        default:
            addChar();
            nextToken = EOF;
            break;
    }

    return nextToken;
} // end of int lookup function

/*****************************************************************************/

/* addChar Function
 *              Adds the next Char to lexeme
 */
void addChar() {
    if (lexLen <= 98) {
        lexeme[lexLen++] = nextChar;
        lexeme[lexLen] = 0;
    }
    else
        printf("Error - lexeme is too long \n");
} // end of addChar function

/*****************************************************************************/

/* getChar Function
 *      Gets the next character of input and determine its character class
 */
void getChar() {
    if ((nextChar = getc(in_fp)) != EOF) {
        if (isalpha(nextChar))
            charClass = LETTER;
        else if (isdigit(nextChar))
            charClass = DIGIT;
        else charClass = UNKNOWN;
    }
    else
        charClass = EOF;
} // end of getChar function

/*****************************************************************************/

/* getNonBlank Function
 *      Calls getChar until it returns a non-whitespace character
 */
void getNonBlank() {
    while (isspace(nextChar))
```

```
        getChar();
} // end of getNonBlank function


/******************************************************************************/

/* lex Function
 *             A simple lexical analyzer for arithmetic expressions
 */
int lex() {
    lexLen = 0;
    getNonBlank();
    switch (charClass) {

        // Parse Identifiers
        case LETTER:
            addChar();
            getChar();
            while (charClass == LETTER || charClass == DIGIT) {
                addChar();
                getChar();
            }
            nextToken = IDENT;
            break;

            // Parse Integer Literals
        case DIGIT:
            addChar();
            getChar();
            while (charClass == DIGIT) {
                addChar();
                getChar();
            }
            nextToken = INT_LIT;
            break;

            // Parentheses and Operators
        case UNKNOWN:
            lookup(nextChar);
            getChar();
            break;

            // EOF
        case EOF:
            nextToken = EOF;
            lexeme[0] = 'E';
            lexeme[1] = 'O';
            lexeme[2] = 'F';
            lexeme[3] = 0;
            break;
    } //end switch

    printf("Next token is: %d, Next lexeme is %s\n", nextToken, lexeme);
    return nextToken;
} //end lex Function

/******************************************************************************/
/*********************************** END ***********************************/
/******************************************************************************/
```

4 Conclusion

I hope that you have found this document somewhat useful. Recall, this is document is meant for novice compiler construction. Furthermore, if you have an issues or concerns with your project please do not hesitate to contact me.

I leave you with one quote to think about on your journey in your compiler construction:

"Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away."

— Antoine de Saint-Exupéry

References

[1] Dr. Christopher Brown. Recursive patterns and context free grammars, 2005.

[2] Instituto Nazionale di Fisica Nucleare. Creating input language analyzers and parsers.

[3] Dr. Frantisek Franek. Compilers, 2002.

[4] Lan Gao. Flex tutorial, 2007.

[5] Technische Universität München. Cup.