KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

Department of Computer Science and Technology

Title: A simple compiler using flex and Bison.

Course No. CSE 3212

Course Title: Compiler Design Laboratory

Submitted by:

Avishek Roy

Roll: ***1807053***

3rd Year 2nd Semester

Department of Computer Science and Engineering

Khulna University of Engineering and Technology, Khulna

Submission date: December 19, 2022

Objective:

After doing this project, we will be able to know

- About Flex and Bison.
- About token and how to declare rules against token.
- How to declare CFG (context free grammar) for different grammar like if else pattern, loop and so on.
- About different patterns and how they work.
- How to create different and new semantic and synthetic rules for the compiler.
- About shift and reduce policy of a compiler.
- About top down and bottom up parser and how they work.

Introduction:

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an *editor*. The file that is created contains what are called the *source statements*. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements.

Flex: Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex. It is a computer program that generates lexical analyzers (also known as "scanners" or "lexers"). An input file describing the lexical analyzer to be generated named *lex.l* is written in lex language. The lex compiler transforms *lex.l* to *C* program, in a file that is always named *lex.yy.c*. The C compiler compiles the *lex.yy.c* file into an executable file called a.out. The output file a.out takes a stream of input characters and produces a stream of tokens.

```
/* definitions */
 ....
%%
/* rules */
....
%%
/* auxiliary routines */
....
```

Bison: GNU Bison, commonly known as Bison, is a parser generator that is part of the GNU Project. Bison reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser (either in C, C++, or Java) that reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. **Bison** command is a replacement for the **yacc**. It is a parser generator similar to *yacc*. Input files should follow the yacc convention of ending in *.y* format.

```
/* definitions */
 ....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```

Instruction in cmd when we use flex and bison together:

1. bison -d hi.y
2. flex hi.l
3. gcc lex.yy.c hi.tab.c -o ex
4. ex

Features of this mini-compiler:

- Import section or header declaration section.
- User define function section.
- Main part or body of the program.
- Declaration of different id or variable and assignment operation.
- If else condition.
- Arithmetic and logical operation.
- Loop (for loop and while loop).
- Switch-case condition.
- Print and show different values.
- Single line and multiple line command.
- Built-in sine function.
- Built-in cosine function.
- Built-in tan function.
- Built-in ln function.
- Built-in log10 function.
- Built-in factorial function.
- Built-in oddeven function.

Token:

A **token** is a pair consisting of a **token** name and an optional attribute value. The **token** name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or sequence of input characters denoting an identifier. The **token** names are the input symbols that the parser processes.

Tokens used in this mini-project:

Bellow in the table it is shown those tokens that is used in this mini-project and their realtine meaning-

Serial no.	Token	Input string	Realtime meaning of Token
1.	NUMBER	"-"?[0-9]+	Any integer number either positive or negative.
2.	VARIABLE	[a-zA_Z0-9]+	Any string using upper case alphabets, lower case alphabets and number.
3.	ENDED	;	Indicate ends of a line.
4.	INT	int	Declaretion of integer variable.
5.	FLOAT	float	Declaretion of floation point variable.
6.	CHAR	char	Declaretion of character variable.
7.	IF	if	If condition start
8.	ELSE	else	Else condition start
9.	FOR	for	For loop line c programming
10.	COLON	:	Colon sign like default
11.	SWITCH	switch	Switch case like c programming
12.	DEFAULT	default	Alternative option
13.	VALUE	value	For printing any variable
14.	ASSIGN	=	Assignment operator
15.	INC	++	Used for increment any value by one
16.	DEC	--	Used for decrement any value by one

17.	LT	<	Less than sign
18.	GT	>	Greater than sign
19.	EQ	===	Check equal or not
20.	PRINT	print	Print something
21.	MAIN	alpha	Start of a program
22.	CASE	case	Different case
23.	Wloop	while	While loop like c program
24.	FACT	fact	Calculating factorial of a number
25.	DEF	def	Funtion defination
26.	HEADER	something.h	Header function
27.	IMPORT	import	Needs for including header function
28.	SIN	sin	Sine function
29.	COS	cos	Cosine function
30.	TAN	tan	Tangent function
31.	LOG	log	Log function
32.	ODD_EVEN	Odd_even	For calculation Oddeven function
33.	‘+’	+	Addition operation
34.	‘-’	-	Subtraction operation
35.	‘*’	*	Multication operation
36.	‘/’	/	Division operation

37.	‘%’	%	Module operation
38.	‘(’	(	First bracket opening
39.	‘)’	)	First bracket closing
40.	‘{’	{	second bracket opening
41.	‘}’	}	second bracket closing
42.	‘^’	^	Power operation
43.	‘[’	[	Third bracket opening
44.	‘]’	]	Third bracket closing
45.	‘,’	,	Comma

Table 1. Realtime meaning of tokens that are used in project

Structure of the project:

CFGs used in this project:

```
program: MAIN '{' statement '}' {printf("\nExecuted successfully\n");}
    ;


type:  FLOAT  {printf("float declaration\n");}
    | INT   {printf("interger declaration\n");}
    | CHAR   {printf("char declaration\n");}
    | CONST  {printf("Constant Declaration\n");}
    ;

id1 : id1 ',' var  {
          int x=$3;
          if(store[x] == 1)
          {
              printf("%c reallocate\n", $3 + 97);
          }
```

```
        else
        {
            store[x] = 1;
        }
        }
    |var    {
        int x=$1;
        printf("Printing variable %d\n", x);
        if(store[x] == 1)
        {
            printf("%c reallocate\n", $1 + 97);
        }
        else
        {
            store[x] = 1;
        }
        }
    ;

statement: ';'  {printf("line ending\n");}
    |declaration    {printf("\n Declaration of the variable\n");}

    |expression ';'         {printf("\nValue of expression: %d\n",$1);$$=$1;}

    | var '=' expression ';'    {
                printf("\nValue of the variable: %d\n",$3);
                int x=$1;
                sym[x]=$3;
                $$=$3;
              }
    |floop '(' NUM ',' NUM ',' NUM ')' BS  statement BE {
                            int i;
                            printf("FOR Loop execution");
                            for(i=$3 ; i<$5 ; i=i+$7 )
                {printf("\nvalue of the  i: %d expression value : %d\n", i,$10);}
                }

    |wloop '(' NUM '<' NUM ')' BS statement BE  {
                            int i;
                            printf("WHILE Loop execution");
                            for(i=$3 ; i<$5 ; i++) {printf("\nvalue of the loop: %d expression value: %d\n", i,$8);}
                }


    | IF '(' expression ')' BS statement BE{
                    if($3){
                        printf("\nvalue of expression in IF: %d\n",$6);
```

```
                }
                else{
                    printf("\ncondition value zero in IF block\n");
                }
                }

| IF '(' expression ')' BS statement BE ELSE BS statement BE{
                        if($3){
                            printf("value of expression in IF: %d\n",$6);
                        }
                        else{
                            printf("value of expression in ELSE: %d\n",$10);
                        }
                        }

| print '(' expression ')' ';' {printf("\nPrint Expression %d\n",$3);}

| factorial '(' NUM ')' ';' {
    printf("\nFACTORIAL declaration\n");
    int i;
    int f=1;
    for(i=1;i<=$3;i++)
    {
        f=f*i;
    }
    printf("FACTORIAL of %d is : %d\n",$3,f);
    }

| odd_even '(' NUM ')' ';' {
    printf("Determining odd or even number \n");

    if($3 % 2 == 0){
        printf("Number : %d is -> Even\n",$3);
    }
    else{
        printf("Number is :%d is -> Odd\n",$3);
    }
    }
| array type var '(' NUM ')' ';' {
    printf("ARRAY Declaration\n");

    printf("Size of the ARRAY is : %d\n",$5);

    }

| function var '(' expression ')' BS statement BE {
    printf("FUNCTION found :  \n");
```

```
    printf("Function Parameter : %d\n",$4);
    printf("Function internal block statement : %d\n",$7);
    }


  | MINMIN

  | MAXMAX
  ;

MINMIN :  minFunc '(' expression ',' expression ')'    { int i=$5;
                        if($3<=$5)   i=$3;
                        printf("\nValue of Min(%d , %d) is : %d\n",$3,$5,i);
                        $$=i;
                            }
                            ;
MAXMAX :   maxFunc '(' expression ',' expression ')'   { int i=$5;
                          if($3>=$5){
                            i=$3;
                          }
                          printf("\nValue of Max(%d , %d) is : %d\n",$3,$5,i);
                          $$=i;
                            }
                            ;

expression: NUM        { printf("\nNumber :  %d\n",$1 ); $$ = $1;  }

  | var          { int x=$1;$$ = sym[x]; }

  | expression '+' expression {printf("\nAddition :%d+%d = %d \n",$1,$3,$1+$3 );  $$ = $1 + $3;}

  | expression '-' expression {printf("\nSubtraction :%d-%d=%d \n ",$1,$3,$1-$3); $$ = $1 - $3; }

  | expression '*' expression {printf("\nMultiplication :%d*%d = %d \n ",$1,$3,$1*$3); $$ = $1 * $3; }

  | expression '||' expression    {printf("\nOR :%d || %d = %d \n ",$1,$3,$1|$3); $$ = $1 * $3; }
  | '~~' expression   {printf("\nNOT :~~ %d = %d \n ",$2,!$2); $$ = !$2; }

  | expression '++'   {printf("\nvalue of %d is incremented and now is %d \n",$1,$1+1 );  $$ = $1 + 1;}

  | expression '--'   {printf("\nvalue of %d is decremented and now is %d \n",$1,$1-1 );  $$ = $1 - 1;}

  | expression '/' expression { if($3){
                      printf("\nDivision :%d/%d \n ",$1,$3,$1/$3);
                $$ = $1 / $3;

                }
```

```
            else{
               $$ = 0;
               printf("\ndivision by zero\n\t");
            }
                    }
| expression '%' expression { if($3){
                    printf("\nMod :%d % %d \n",$1,$3,$1 % $3);
                    $$ = $1 % $3;

                  }
                 else{
                  $$ = 0;
                  printf("\nMOD by zero\n");
                  }
                    }
| expression '^' expression {printf("\nPower  :%d ^ %d \n",$1,$3);  $$ = pow($1 , $3);}

| expression '<' expression {printf("\nLess Than :%d < %d \n",$1,$3); $$ = $1 < $3 ; }
| expression '>' expression {printf("\nGreater than :%d > %d \n ",$1,$3); $$ = $1 > $3; }


| expression '!!!' expression   {printf("\ninequality :%d !!! %d \n ",$1,$3);
                          if($1!=$3){
                           $$ = $1 != $3 ;
                          }
                           else{
                              $$ = $1 == $3 ;
                           }
                            }

| expression '===' expression   {printf("\nequality :%d === %d \n ",$1,$3);   if($1==$3) $$ = $1==$3 ;   else $$ = $1!=$3; }


| SIN expression          {printf("\nValue of Sin(%d) is : %lf\n",$2,sin($2*3.1416/180));
                        $$=sin($2*3.1416/180);
                        }


|SQUARE '(' expression ')' {
                        printf("\nSquare of %d is : %d\n",$3,$3 * $3);
                        $$ = $3 * $3;
                      }
|CUBIC '(' expression ')' {
                        printf("\nCube of %d is : %d\n",$3,$3 * $3 * $3);
                        $$ = $3 * $3 * $3;
                      }
```

```
| COS expression          {printf("\nValue of Cos(%d) is : %lf\n",$2,cos($2*3.1416/180)); $$=cos($2*3.1416/180);}

| TAN expression          {printf("\nValue of Tan(%d) is : %lf\n",$2,tan($2*3.1416/180)); $$=tan($2*3.1416/180);}

| LOG expression          {printf("\nValue of Log(%d) is : %lf\n",$2,(log($2))); $$=(log($2));}

;

declaration: type id1 {printf("\nvariable Declaration\n");}
    ;
```

Discussion:

The input code is parsed using a bottom-up parser in this compiler. Because it is only built with flex and bison, this compiler is unable to provide original functionality such as if-else, loop, and switch case features. However, when creating code in this compiler-specific style, header declaration is not required but if we need we can use header file. The float variable always returns a value in the double data type, which is a compiler requirement. Any variable's string value is not stored by this compiler. With certain modifications, the code format supported by this compiler is similar to that of the C language. This compiler is error-free while working with the stated CFG format.

Conclusion:

Every programming language has required the use of a compiler. Designing a new language without a solid understanding of how a compiler works may be a challenging endeavor. Several issues were encountered during the design phase of this compiler, such as loop, if-else, switch case functions not working as they should owing to bison limitations, character and string variable values not being stored properly, and so on. In the end, some of these issues were resolved, and given the constraints, this compiler performs admirably.

References:

- https://whatis.techtarget.com/definition/compiler
- Principles of Compiler Design By Alfred V.Aho & J.D Ullman