KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

Department of Computer Science and Engineering

Project Report on CSE 3212

Compiler Design Laboratory

Date of Submission: 21 November, 2023

Submitted By	**Submitted To**
Sagar Dutta Roll: 1907085 Year: 3rd Semester: 2nd Department of Computer Science and Engineering (CSE) Khulna University of Engineering and Technology (KUET), Khulna-9203	Nazia Jahan Khan Chowdhury Assistant Professor Dipannita Biswas Lecturer Department of Computer Science and Engineering (CSE) Khulna University of Engineering and Technology (KUET), Khulna-9203

Objective:

The main purposes of this lab are given below:

- To gather knowledge about compiler and how a compiler function.
- To learn bison and flex
- To learn how to generate token from a language
- To create a new programming language that is similar to the basic C programming language

Introduction:

Compilers serve as the backbone of modern software development, translating high-level programming languages into executable code. In this project, the focus was on constructing a compiler using the powerful tools Flex and Bison. This report details the journey undertaken to build a functional compiler, outlining the rationale behind the selection of Flex and Bison, as well as the methodologies employed in the various stages of development.

Flex:

Flex is a tool that generates token. It also does the job of pattern matching. It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program. The extension of flex file is '.l'. It creates a C file named lex.yy.c after execution. A Flex file contains yylex() function which starts scanning and called yywrap(). When the scanning is oven then yywrap() returns and the scanning is finished. Structure of a file that is compatible with Flex is as follows: {Definitions}

%%

{Rules}

%%

{User subroutine}

Bison:

Bison is a parser generator. The flex file provides token or scanners to a bison file and the bison file generates a syntax tree. The file extension for a bison file is '.y' (yacc). YACC- Yet Another Compiler Compiler is used to generate a '.h' and '.c' file. The '.h' file contains the definitions of the tokens the '.l' file returns. The function "yyparse()" is called by yacc which calls the yylex() function to get the tokens. Bison stores the parsed tokens in two stacks: parse stack, and value stack. In value stack the values of the parsed tokens are assigned automatically starting from ASCII value 258. The parsed tree generated by Bison is calculated from bottom to top. Input file structure of a bison file is as follows:

```
%}

C declarations

%}

Bison Declarations

%%

Grammar Rules Declarations in BNF form (CFG)

%%

Additional C codes
```

Workflow of Flex and Bison:

The following diagram illustrates the workflow of flex and bison:

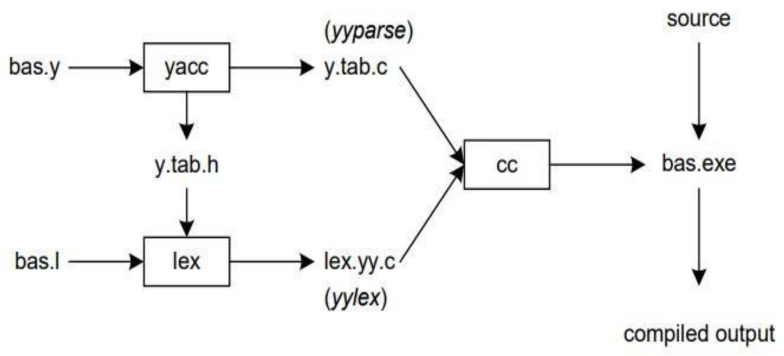

Figure 2: Building a Compiler with Lex/Yacc

Running Bison and Flex Program:

The command prompt script for running bison and flex program is given step by step:

- bison -d file_name.y
- flex file_name.l
- gcc lex.yy.c file_name.tab.c -o obj
- obj

Compiler Overview:

This manual compiler consists of two major components. Which are Tokens and Context free grammar (CFG). Their description is given below:

Tokens:

A token is the smallest element(character) of a computer language program that is meaningful to the compiler. The parser has to recognize these as tokens: identifiers, keywords, literals, operators, punctuators, and others separator.

My custom tokens are:

- SCAN: This token is returned to parser user will put an input.
- ASGN: This token is returned to parser when it matches “assign_f”
- SHOW_VAR: This token is returned to parser when it matches “show_var”
- SHOW_STR: This token is returned to parser when it matches “show_str”
- SHOW_LINE: This token is returned to parser when it matches “show_line”
- INT: This token is returned to parser when it matches “intgs”
- FLOAT: This token is returned to parser when it matches “floats”
- DOUBLE: This token is returned to parser when it matches “doubles”
- CHAR: This token is returned to parser when it matches “chars”
- FBS: First bracket matches starts
- FBE: First bracket matches ends
- SBS: Second bracket matches starts
- SBE: Second bracket matches ends
- VARIABLE: This token is returned to parser when it matches any characters
- MAIN: This token is returned to parser when it matches “mains”
- DEFINING: This token is returned to parser when it matches “defining”
- PRINTING: This token is returned to parser when it matches “printing”
- INCREMENT_By: This token is returned to parser when it matches “inc_by”
- IF: This token is returned to parser when it matches “if_f”.
- ELSEIF: This token is returned to parser when it matches “again_if_f”.
- ELSE: This token is returned to parser when it matches “else_f”.
- FOR: This token is returned to parser when it matches “for_loop”
- TO: This token is used to return to parser when it matches “TO”
- WHILE: This token is returned to parser when it matches “while”

- SWITCH: This token is returned to parser when it matches “switch_to”.
- SWITCH _TO_CASE: This token is returned to parser when it matches “switch_to_case”.
- FUNCTION: This token is returned to parser when it matches “func_”.
- MODULUS: This token is returned to parser when it matches “modulus”
- DEFAULT: This token is returned to parser when it matches “assign_f”
- SEMICOLON: This token is returned to parser when it matches “;”
- COLON: This token is returned to parser when it matches “:”
- COMMA: This token is returned to parser when it matches “,”
- PLUS: This token is returned to parser when it matched “add”
- MINUS: This token is returned to parser when it matched “sub“
- MUL: This token is returned to parser when it matched “multi”
- DIV: This token is returned to parser when it matched “div”
- POWER: This token is returned to parser when it matched “power”
- MOD: This token is returned to parser when it matches “mod”
- DEFAULT: This token is returned to parser when it matches “default”
- EQ: This token is returned to parser when it matches “equal_to”
- NQ: This token is returned to parser when it matches “not_equal_to”
- LESS: This token is returned to parser when it matches “Less”
- GREAT: This token is returned to parser when it matches “Great”
- LE: This token is returned to parser when it matches “less_than_equal”
- GE: This token is returned to parser when it matches “greater_than_equal”

Context Free Grammar (CFG):

Context-free grammars (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but they cannot describe all possible languages.

My custom CFGs are:

```
starthere        : function program function ;

program    : INT MAIN FBS FBE SBS statement SBE { printf("\nCompilation
Successful\n"); } ;

statement        : /* empty */

            | statement declaration| statement print| statement expression| statement
            ifelse | statement assign| statement forloop| statement switch| statement
            whileloop ;


/*--------DECLARATION STARTS-------*/

declaration : type variables SEMICOLON {}

type             : INT | DOUBLE | CHAR {};

variables        : variable COMMA variables {} | variable {};

variable         : ID

{

int x = addnewval($1,0);

                 if(!x) {

                          printf("Error:Variable %s is already declared\n",$1);

                                   exit(-1);}}

                          | ID ASGN expression

                                            {int x = addnewval($1,$3);

                                   if(!x) {

                                   printf("Error: Variable %s is already declared\n",$1);

                                   exit(-1);}};

/*-------DECLARATION ENDS----------*/

/*------VARIABLE ASSIGN STARTS-----*/
```

```
assign : ID ASGN expression SEMICOLON

                    {if(!isdeclared($1)) {

                              printf("Error: Variable %s is not declared\n",$1);

                              exit(-1); }

                              else{setval($1,$3); } }

                    | SCAN FBS ID FBE SEMICOLON

                    {int tmp;

                              tmp = 7;

                              scanf("%d", &tmp);

                              setval($3, tmp);};

/*------VARIABLE ASSIGN ENDS-------*/

/*--------PRIINTF STARTS----------*/

print               : SHOW_VAR FBS ID FBE SEMICOLON

                    {if(!isdeclared($3)){

                    printf("Error: Variable %s is not declared\n",$3);

                              exit(-1)}

                              else{

                              int v = getval($3);

                              printf("%d",v);}}

                              | SHOW_STR FBS STR FBE SEMICOLON

                              {int l = strlen($3);

                              int i;

                              for(i = 1;  i < l-1; i++) printf("%c",$3[i]);}

                              | SHOW_LINE FBS FBE   SEMICOLON

                              {printf("\n");};

/*--------PRINTF ENDS------------*/

/*--------EXPRESSION STARTS--------*/

expression : NUM {$$ = $1;}
```

```
| ID
{
        if(!isdeclared($1)) {
        printf("Error : Variable %s is not declared\n",$1);
                exit(-1);
                        }
        else{
                $$ = getval($1);
                }
}
| expression POWER expression
                {$$ = pow($1,$3);}
| expression MODULUS expression
                {$$ = $1 % $3;}
| expression PLUS expression
                {$$ = $1 + $3;}
| expression MINUS expression
                {$$ = $1 - $3;}
| expression MUL expression
                {$$ = $1 * $3;}
| expression DIV expression
                {
                        if($3) {$$ = $1 / $3;}
                        else {
                                $$ = 0;
                printf("\nMath Error: Division by zero\t");
                                exit(-1);
                        }
```

```
                    }
            | expression LT expression
                    { $$ = $1 < $3; }
            | expression GT expression
                    { $$ = $1 > $3; }
            | expression LE expression
                    { $$ = $1 <= $3; }
            | expression GE expression
                    { $$ = $1 >= $3; }
            | expression NE expression
                    { $$ = $1 != $3;}
            | expression EQ expression
                    { $$ = $1 == $3; }
            | FBS expression FBE
                    {$$ = $2;}
            ;
/*--------EXPRESSION ENDS--------*/
/*---------IF ELSE STARTS---------*/
ifelse  : IF FBS ifexp FBE SBS LoopStatement SBE elseif
                    {if($3)
                        printf("%s", $6);
if_done_list[if_pointer] = 0;
                        if_pointer--;};
ifexp   : expression
                    {if_pointer++;
                        if_done_list[if_pointer] = 0;
                        if($1){
                            printf("\nIf executed\n");
```

```
                                                            if_done_list[if_pointer] = 1;}
                                                    $$ = $1;};
elseif    : /* empty */| elseif ELSEIF FBS expression FBE SBS LoopStatement SBE
                                              {if($4 && if_done_list[if_pointer] == 0){
                                                            printf("%s", $7);
                          if_done_list[if_pointer] = 1;}}
                    | elseif ELSE SBS LoopStatement SBE
                                              {
                                    if(if_done_list[if_pointer] == 0){
printf("%s", $4);
                                                            if_done_list[if_pointer] = 1;
                                                    }
                                              }


                    ;

/*---------IF ELSE ENDS------------*/




/*--------WHILE START-------------*/

whileloop : WHILE FBS ID LESS NUM FBE SBS LoopStatement SBE
                    {
                          int tmp = getval($3);

                          while(tmp<$5)
                          {
```

```
                    printf("%d ", tmp);

                    printf("%s",$8);

                    tmp = tmp+1;

               }

               setval($3, tmp);


          }

          | WHILE FBS ID GREAT NUM FBE SBS LoopStatement SBE

          {

               int tmp = getval($3);

               while(tmp>$5)

               {

                    printf("%d ", tmp);

                    printf("%s",$8);

                    tmp = tmp-1;

               }

               setval($3, tmp);

          }

          ;

/*--------WHILW ENDS------------*/

/*------FOR LOOP STARTS----------*/

forloop: FOR FBS expression TO expression INCREMENT_BY expression FBE SBS
LoopStatement SBE

                         {

                              int st = $3;

                              int ed = $5;

                              int dif = $7;

                              int cnt = 0;
```

```
                                                    int k = 0;

                        for(k = st; k <= ed; k += dif){
                                                    cnt++;
                                                         int r ;
                                                         if(strlen($10)!=0)
                                                         printf("%s\n", $10);}
                                          printf("Loop executes %d times\n",cnt);};
                   LoopStatement: { strcpy($$,"");}
                                     | LoopStatement Lprint {strcat($$, $2);};
                   Lprint            : SHOW_VAR FBS ID FBE SEMICOLON
                                                {char val[1000];
                                                    strcpy(val, "");
                                                    if(!isdeclared($3)){
                                     strcat(val, "Compilation Error: Variable ");
                                                         char tmp[20];
                                                         sprintf(tmp, "%s ", $3);
                                                         strcat(val, tmp);
                                                         strcat( val,"is not declared\n");
                                                         printf("%s", val);
                                                         exit(-1);
                                                         strcpy($$, val);
                                                    }
                        else{char val[1000];
                        strcpy(val, "");
                        int v = getval($3);
                        char tmp[18];
                        sprintf(tmp, "%d", v);
```

```
          strcat(val, tmp);
          strcpy($$, val);}}
                              | SHOW_STR FBS STR FBE SEMICOLON
                                   {char val[10000];
                                   strcpy(val, "");
                                   int l = strlen($3);
                                   int i;
                                   for(i = 1;  i < l-1; i++)
                                   {char tmp[20];
                                   sprintf(tmp, "%c", $3[i]);
                                    strcat(val, tmp);}
                                   strcpy($$, val);}
                              | SHOW_LINE FBS FBE   SEMICOLON
                              {strcpy($$, "\n");};
/*------FOR LOOP ENDS------------*/
/*------SWITCH CASE STARTS--------*/
switch  : SWITCH FBS expswitch FBE SBS switchinside SBE ;
expswitch        :  expression
                 {switch_complete = 0;switch_variable = $1;};
switchinside     : /* empty */| switchinside expression COLON SBS LoopStatement SBE
                                        {if($2 == switch_variable){
                                                  //printf("Executed %d\n",$2);
                                                  printf("%s", $5);
                                                  switch_complete = 1;
                                             }}
| switchinside DEFAULT COLON SBS LoopStatement SBE
                                        {if(switch_complete == 0){
                                                  switch_complete = 1;
```

```
                                                printf("Default Block executed\n");
                                                     printf("%s", $5);}};
/*------SWTICH CASE ENDS----------*/
/*------FUNCTION  STARTS-----------*/
function          : /* empty */| function func;
func     : type FUNCTION FBS fparameter FBE SBS statement SBE
                                           {printf("Function Declared\n");};
fparameter      : /* empty */| type ID fsparameter;
fsparameter : /* empty */| fsparameter COMMA type ID;
/*-------FUNCTION ENDS------------*/
```

Discussion:

The development of this compiler using Flex and Bison revealed both successes and challenges in the implementation process. The synergy between Flex's lexical analysis capabilities and Bison's parsing functionalities significantly streamlined the initial stages of language interpretation. However, complexities arose during the semantic analysis phase, demanding meticulous handling of grammar rules . The project's success in generating executable code underscored the robustness of the implemented algorithms. Furthermore, the comparison with established compilers highlighted areas of parity and divergence, illuminating potential avenues for improvement. Addressing these intricacies not only enhanced power of compiler theory but also underscored the practical considerations essential for producing efficient and reliable language translators.

Conclusion:

Designing a new programming language necessitates a profound understanding of compilers, as they form the backbone of language functionality. This project encountered numerous challenges during the compiler's design phase, including unexpected limitations within Bison causing issues with loops, conditional statements like 'if-else' and 'switch-case,' and functions. Furthermore, hurdles emerged regarding the proper storage of character and string variable values. Despite these impediments, diligent efforts were made to rectify several of these issues by the project's conclusion. Despite the inherent limitations, the compiler now performs satisfactorily, considering the constraints it operates within.

References:

- Flex, version 2.5.4 by Vern Paxson
- Bison, Version 2.4.1 by Robert Corbett and Richard Stallman
- LEX & YACC Tutorial by Tom Niemann