Parser for the C Language

National Institute of Technology Karnataka, Surathkal

Date: 24th February 2020

Submitted To: Dr. P. Santhi Thilagam

Group members:

1. Aditya Rastogi 171CO105
2. Animesh Kumar 171CO108
3. Manas Gupta 171CO122

Abstract

This report contains the details of the tasks finished as a part of Phase Two of the Compiler Design Lab. This phase involved the development of a Parser for C language which makes use of the C lexer to parse the given C input file. The previous work was concentrated at developing a lexical analyzer(flex script) to generate a stream of tokens from the source code and populate the symbol table. The lexical analyzer is only able to handle errors involving invalid tokens. However, in C language there may also be present number of errors in the structure of a language (syntax), unbalanced parenthesis etc. Thus a parser is needed to be developed for handling these errors. After the lexical phase, the compiler enters the syntax analysis phase. Thus the parse becomes a part of the syntax analysis phase. 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. This is done by a parser. The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

Contents

List of Figures and Tables

Introduction

Syntactic Analysis and Parser

The lexical analyzer (flex scripts) returns a stream of tokens, which populates the symbol table. These tokens are then taken as input by the Parser. Parser verifies that a string of token names can be generated by the grammar of the source language. The Parser then reports any syntax errors in an intelligible manner and recovers from the commonly occurring errors to continue processing the remainder of the program. The Parser recognizes the following types of errors.

1. Structural errors

2. Missing identifiers

3. Wrong keywords

4. Unbalanced parenthesis

Parser is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer. Parser is mainly classified into 2 categories; Top-down Parser, and Bottom-up Parser. These are explained as follows:

1. Top-down Parser:

Top-down Parser is the Parser, which generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e., it starts from the start symbol and ends on the terminals.

2. Bottom-up Parser:

Bottom-up Parser is the Parser that generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e., it starts from non-terminals and ends on the stat symbol. It uses the reverse of the rightmost derivation.

YACC Script

Yacc is written in a portable dialect of C, and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C. Yacc provides a comprehensive tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then

generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then the user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions. The lexer can be used to make a simple parser. But it needs making extensive use of the user-defined states. The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the wrong data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the wrong data. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules. The structure of our Yacc script is given below; a full specification file looks like,

```
declarations

%%

rules

%%

programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also.

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A  :  BODY  ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation. Each rule can have an associated action, which is executed after all the component symbols of the rule have been parsed. Actions are basically C-program statements

surrounded by curly braces.

C Program

This section describes the input C program which is fed to the yacc script for parsing. The workflow is explained as under:

1. Compile Yacc script by this command

```
$ yacc -d c_parser.y54
```

2. Compile the flex script using Flex tool

```
$ flex c_lexer.l
```

3. After compiling the lex file, a lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.

4. The three files, lex.yy.c, y.tab.c and y.tab.h are compiled together with the options –ll and –ly

```
$ gcc -o compiler lex.yy.c y.tab.h y.tab.c -ll -ly 5.
```

The executable file is generated, which on running parses the C file given as a command line input

```
$ ./compiler test.c
```

The script also has an option to take standard input instead of taking input from a file.

Design of Programs

Code:

Updated Lexer Code:

```
%{
   #include <stdio.h>
   #include <string.h>
   #include "y.tab.h"

   #define ANSI_COLOR_RED      "\x1b[31m"
   #define ANSI_COLOR_GREEN    "\x1b[32m"
   #define ANSI_COLOR_YELLOW   "\x1b[33m"
   #define ANSI_COLOR_BLUE     "\x1b[34m"
   #define ANSI_COLOR_MAGENTA  "\x1b[35m"
   #define ANSI_COLOR_CYAN     "\x1b[36m"
   #define ANSI_COLOR_RESET    "\x1b[0m"

   struct symboltable
   {
       char name[100];
       char class[100];
       char type[100];
       char value[100];
```

```
    int lineno;
    int length;
}ST[1007];


struct constanttable
{
    char name[100];
    char type[100];
    int length;
}CT[1007];


int hash(char *str)
{
    int value = 0;
    for(int i = 0 ; i < strlen(str) ; i++)
    {
        value = 10*value + (str[i] - 'A');
        value = value % 1007;
        while(value < 0)
            value = value + 1007;
    }
    return value;
}
```

```
int lookupST(char *str)

{

    int value = hash(str);

    if(ST[value].length == 0)

    {

        return 0;

    }

    else if(strcmp(ST[value].name,str)==0)

    {

        return 1;

    }

    else

    {

        for(int i = value + 1 ; i!=value ; i = (i+1)%1007)

        {

            if(strcmp(ST[i].name,str)==0)

            {

                return 1;

            }

        }

        return 0;

    }

}
```

```
int lookupCT(char *str)
{
    int value = hash(str);
    if(CT[value].length == 0)
        return 0;
    else if(strcmp(CT[value].name,str)==0)
        return 1;
    else
    {
        for(int i = value + 1 ; i!=value ; i = (i+1)%1007)
        {
            if(strcmp(CT[i].name,str)==0)
            {
                return 1;
            }
        }
        return 0;
    }
}


void insertST(char *str1, char *str2)
{
    if(lookupST(str1))
    {
```

```
            return;
        }
        else
        {
            int value = hash(str1);
            if(ST[value].length == 0)
            {
                strcpy(ST[value].name,str1);
                strcpy(ST[value].class,str2);
                ST[value].length = strlen(str1);
                insertSTline(str1,yylineno);
                return;
            }

            int pos = 0;

            for (int i = value + 1 ; i!=value ; i = (i+1)%1007)
            {
                if(ST[i].length == 0)
                {
                    pos = i;
                    break;
                }
            }
```

```
            strcpy(ST[pos].name,str1);
            strcpy(ST[pos].class,str2);
            ST[pos].length = strlen(str1);
        }
    }

void insertSTtype(char *str1, char *str2)
{
    for(int i = 0 ; i < 1007 ; i++)
    {
        if(strcmp(ST[i].name,str1)==0)
        {
            strcpy(ST[i].type,str2);
        }
    }
}

void insertSTvalue(char *str1, char *str2)
{
    for(int i = 0 ; i < 1007 ; i++)
    {
        if(strcmp(ST[i].name,str1)==0)
        {
```

```
            strcpy(ST[i].value,str2);
        }
    }
}

void insertSTline(char *str1, int line)
{
    for(int i = 0 ; i < 1007 ; i++)
    {
        if(strcmp(ST[i].name,str1)==0)
        {
            ST[i].lineno = line;
        }
    }
}

void insertCT(char *str1, char *str2)
{
    if(lookupCT(str1))
        return;
    else
    {
        int value = hash(str1);
        if(CT[value].length == 0)
```

```
            {
                strcpy(CT[value].name,str1);
                strcpy(CT[value].type,str2);
                CT[value].length = strlen(str1);
                return;
            }

            int pos = 0;

            for (int i = value + 1 ; i!=value ; i = (i+1)%1007)
            {
                if(CT[i].length == 0)
                {
                    pos = i;
                    break;
                }
            }

            strcpy(CT[pos].name,str1);
            strcpy(CT[pos].type,str2);
            CT[pos].length = strlen(str1);
        }
    }
```

```
   void printST()

   {

       printf("%10s | %15s | %10s | %10s | %10s\n","SYMBOL", "CLASS",
"TYPE","VALUE", "LINE NO");

       for(int i=0;i<81;i++) {

           printf("-");

       }

       printf("\n");

       for(int i = 0 ; i < 1007 ; i++)

       {

           if(ST[i].length == 0)

           {

               continue;

           }

           printf("%10s | %15s | %10s | %10s | %10d\n",ST[i].name,
ST[i].class, ST[i].type, ST[i].value, ST[i].lineno);

       }

   }




   void printCT()

   {

       printf("%10s | %15s\n","NAME", "TYPE");

       for(int i=0;i<81;i++) {
```

```
            printf("-");

        }

        printf("\n");

        for(int i = 0 ; i < 1007 ; i++)

        {

            if(CT[i].length == 0)

                continue;


            printf("%10s | %15s\n",CT[i].name, CT[i].type);

        }

    }

    char curid[20];

    char curtype[20];

    char curval[20];


%}


DE "define"

IN "include"


%%

\n  {yylineno++;}

([#]["  "]*({IN})[  ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|"
"|"\t"]   { }
```

```
([#][" "]*({DE})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"]
{ }

\/\/(.*)
{ }

\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/
{ }

[ \n\t] ;

";"             { return(';'); }

","             { return(','); }

("{")           { return('{'); }

("}")           { return('}'); }

"("             { return('('); }

")"             { return(')'); }

("["|"<:")      { return('['); }

("]"|":>")      { return(']'); }

":"             { return(':'); }

"."             { return('.'); }



"char"          { strcpy(curtype,yytext); insertST(yytext, "Keyword");
return CHAR;}

"double"        { strcpy(curtype,yytext); insertST(yytext, "Keyword");
return DOUBLE;}

"else"          { insertSTline(yytext, yylineno); insertST(yytext,
"Keyword"); return ELSE;}

"float"         { strcpy(curtype,yytext); insertST(yytext,
"Keyword");return FLOAT;}
```

```
"while"          { insertST(yytext, "Keyword"); return WHILE;}

"do"             { insertST(yytext, "Keyword"); return DO;}

"for"            { insertST(yytext, "Keyword"); return FOR;}

"if"             { insertST(yytext, "Keyword"); return IF;}

"int"            { strcpy(curtype,yytext); insertST(yytext,
"Keyword");return INT;}

"long"           { strcpy(curtype,yytext); insertST(yytext, "Keyword");
return LONG;}

"return"         { insertST(yytext, "Keyword");  return RETURN;}

"short"          { strcpy(curtype,yytext); insertST(yytext, "Keyword");
return SHORT;}

"signed"         { strcpy(curtype,yytext); insertST(yytext, "Keyword");
return SIGNED;}

"sizeof"         { insertST(yytext, "Keyword");  return SIZEOF;}

"struct"         { strcpy(curtype,yytext); insertST(yytext, "Keyword");
return STRUCT;}

"unsigned"       { insertST(yytext, "Keyword");  return UNSIGNED;}

"void"           { strcpy(curtype,yytext); insertST(yytext, "Keyword");
return VOID;}

"break"          { insertST(yytext, "Keyword");  return BREAK;}



"++"             { return increment_operator; }

"--"             { return decrement_operator; }

"<<"             { return leftshift_operator; }

">>"             { return rightshift_operator; }
```

```
"<="            { return lessthan_assignment_operator; }

"<"             { return lessthan_operator; }

">="            { return greaterthan_assignment_operator; }

">"             { return greaterthan_operator; }

"=="            { return equality_operator; }

"!="            { return inequality_operator; }

"&&"            { return AND_operator; }

"||"            { return OR_operator; }

"^"             { return caret_operator; }

"*="            { return multiplication_assignment_operator; }

"/="            { return division_assignment_operator; }

"%="            { return modulo_assignment_operator; }

"+="            { return addition_assignment_operator; }

"-="            { return subtraction_assignment_operator; }

"<<="           { return leftshift_assignment_operator; }

">>="           { return rightshift_assignment_operator; }

"&="            { return AND_assignment_operator; }

"^="            { return XOR_assignment_operator; }

"|="            { return OR_assignment_operator; }

"&"             { return amp_operator; }

"!"             { return exclamation_operator; }

"~"             { return tilde_operator; }

"-"             { return subtract_operator; }

"+"             { return add_operator; }
```

```
"*"             { return multiplication_operator; }

"/"             { return division_operator; }

"%"             { return modulo_operator; }

"|"             { return pipe_operator; }

\=              { return assignment_operator;}


\"[^\n]*\"/[;|,|\)]             {strcpy(curval,yytext);
insertCT(yytext,"String Constant"); return string_constant;}

\'[A-Z|a-z]\'/[;|,|\)|:]        {strcpy(curval,yytext);
insertCT(yytext,"Character Constant"); return character_constant;}

[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[  {strcpy(curid,yytext); insertST(yytext,
"Array Identifier");  return identifier;}

[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]
{strcpy(curval,yytext); insertCT(yytext, "Number Constant"); return
integer_constant;}

([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^]
{strcpy(curval,yytext); insertCT(yytext, "Floating Constant"); return
float_constant;}

[A-Za-z_][A-Za-z_0-9]*
{strcpy(curid,yytext);insertST(yytext,"Identifier");  return identifier;}


(.?) {

       if(yytext[0]=='#')

       {

           printf("Error in Pre-Processor directive at line no.
%d\n",yylineno);

       }
```

```
        else if(yytext[0]=='/')
        {
            printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);
        }
        else if(yytext[0]=='"')
        {
            printf("ERR_INCOMPLETE_STRING at line no. %d\n",yylineno);
        }
        else
        {
            printf("ERROR at line no. %d\n",yylineno);
        }
        printf("%s\n", yytext);
        return 0;
}


%%
```

Parser Code:

```
%{

   void yyerror(char* s);

   int yylex();

   #include "stdio.h"

   #include "stdlib.h"

   #include "ctype.h"

   #include "string.h"

   void ins();

   void insV();

   int flag=0;


   #define ANSI_COLOR_RED      "\x1b[31m"

   #define ANSI_COLOR_GREEN    "\x1b[32m"

   #define ANSI_COLOR_CYAN     "\x1b[36m"

   #define ANSI_COLOR_RESET    "\x1b[0m"


   extern char curid[20];

   extern char curtype[20];

   extern char curval[20];


%}
```

```
%nonassoc IF

%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT

%token RETURN MAIN

%token VOID

%token WHILE FOR DO

%token BREAK

%token ENDIF

%expect 2


%token identifier

%token integer_constant string_constant float_constant character_constant


%nonassoc ELSE


%right leftshift_assignment_operator rightshift_assignment_operator

%right XOR_assignment_operator OR_assignment_operator

%right AND_assignment_operator modulo_assignment_operator

%right multiplication_assignment_operator division_assignment_operator

%right addition_assignment_operator subtraction_assignment_operator

%right assignment_operator


%left OR_operator

%left AND_operator

%left pipe_operator
```

```
%left caret_operator

%left amp_operator

%left equality_operator inequality_operator

%left lessthan_assignment_operator lessthan_operator
greaterthan_assignment_operator greaterthan_operator

%left leftshift_operator rightshift_operator

%left add_operator subtract_operator

%left multiplication_operator division_operator modulo_operator



%right SIZEOF

%right tilde_operator exclamation_operator

%left increment_operator decrement_operator




%start program


%%

program

           : declaration_list;


declaration_list

           : declaration D


D
```

```
            : declaration_list
            | ;

declaration
            : variable_declaration
            | function_declaration
            | structure_definition;

variable_declaration
            : type_specifier variable_declaration_list ';'
            | structure_declaration;

variable_declaration_list
            : variable_declaration_identifier V;

V
            : ',' variable_declaration_list
            | ;

variable_declaration_identifier
            : identifier { ins(); } vdi;

vdi : identifier_array_type | assignment_operator expression ;
```

```
identifier_array_type

            : '[' initilization_params

            | ;


initilization_params

            : integer_constant ']' initilization

            | ']' string_initilization;


initilization

            : string_initilization

            | array_initialization

            | ;


type_specifier

            : INT | CHAR | FLOAT | DOUBLE

            | LONG long_grammar

            | SHORT short_grammar

            | UNSIGNED unsigned_grammar

            | SIGNED signed_grammar

            | VOID ;


unsigned_grammar

            : INT | LONG long_grammar | SHORT short_grammar | ;
```

```
signed_grammar

            : INT | LONG long_grammar | SHORT short_grammar | ;



long_grammar

            : INT | ;



short_grammar

            : INT | ;



structure_definition

            : STRUCT identifier { ins(); } '{' V1  '}' ';';



V1 : variable_declaration V1 | ;



structure_declaration

            : STRUCT identifier variable_declaration_list;




function_declaration

            : function_declaration_type
function_declaration_param_statement;



function_declaration_type

            : type_specifier identifier '('  { ins();};
```

```
function_declaration_param_statement
            : params ')' statement;


params
            : parameters_list | ;


parameters_list
            : type_specifier parameters_identifier_list;


parameters_identifier_list
            : param_identifier parameters_identifier_list_breakup;


parameters_identifier_list_breakup
            : ',' parameters_list
            | ;


param_identifier
            : identifier { ins(); } param_identifier_breakup;


param_identifier_breakup
            : '[' ']'
            | ;
```

```
statement

            : expression_statment | compound_statement

            | conditional_statements | iterative_statements

            | return_statement | break_statement

            | variable_declaration;


compound_statement

            : '{' statment_list '}' ;


statment_list

            : statement statment_list

            | ;


expression_statment

            : expression ';'

            | ';' ;


conditional_statements

            : IF '(' simple_expression ')' statement
conditional_statements_breakup;


conditional_statements_breakup

            : ELSE statement

            | ;
```

```
iterative_statements
           : WHILE '(' simple_expression ')' statement
           | FOR '(' expression ';' simple_expression ';' expression ')'
           | DO statement WHILE '(' simple_expression ')' ';';

return_statement
           : RETURN return_statement_breakup;

return_statement_breakup
           : ';'
           | expression ';' ;

break_statement
           : BREAK ';' ;

string_initilization
           : assignment_operator string_constant { insV(); };

array_initialization
           : assignment_operator '{' array_int_declarations '}';

array_int_declarations
           : integer_constant array_int_declarations_breakup;
```

```
array_int_declarations_breakup
            : ',' array_int_declarations
            | ;

expression
            : mutable expression_breakup
            | simple_expression ;

expression_breakup
            : assignment_operator expression
            | addition_assignment_operator expression
            | subtraction_assignment_operator expression
            | multiplication_assignment_operator expression
            | division_assignment_operator expression
            | modulo_assignment_operator expression
            | increment_operator
            | decrement_operator ;

simple_expression
            : and_expression simple_expression_breakup;

simple_expression_breakup
            : OR_operator and_expression simple_expression_breakup | ;
```

```
and_expression
            : unary_relation_expression and_expression_breakup;

and_expression_breakup
            : AND_operator unary_relation_expression and_expression_breakup
            | ;

unary_relation_expression
            : exclamation_operator unary_relation_expression
            | regular_expression ;

regular_expression
            : sum_expression regular_expression_breakup;

regular_expression_breakup
            : relational_operators sum_expression
            | ;

relational_operators
            : greaterthan_assignment_operator |
lessthan_assignment_operator | greaterthan_operator
            | lessthan_operator | equality_operator | inequality_operator ;
```

```
sum_expression

            : sum_expression sum_operators term

            | term ;


sum_operators

            : add_operator

            | subtract_operator ;


term

            : term MULOP factor

            | factor ;


MULOP

            : multiplication_operator | division_operator | modulo_operator
;


factor

            : immutable | mutable ;


mutable

            : identifier

            | mutable mutable_breakup;


mutable_breakup
```

```
            : '[' expression ']'
            | '.' identifier;

immutable
            : '(' expression ')'
            | call | constant;

call
            : identifier '(' arguments ')';

arguments
            : arguments_list | ;

arguments_list
            : expression A;

A
            : ',' expression A
            | ;

constant
            : integer_constant  { insV(); }
            | string_constant   { insV(); }
            | float_constant    { insV(); }
```

```
            | character_constant{ insV(); };


%%


extern FILE *yyin;
extern int yylineno;
extern char *yytext;
void insertSTtype(char *,char *);
void insertSTvalue(char *, char *);
void incertCT(char *, char *);
void printST();
void printCT();


int main(int argc , char **argv)
{
   yyin = fopen(argv[1], "r");
   yyparse();


   if(flag == 0)
   {
       printf(ANSI_COLOR_GREEN "Status: Parsing Complete - Valid"
ANSI_COLOR_RESET "\n");
       printf("%30s" ANSI_COLOR_CYAN "SYMBOL TABLE" ANSI_COLOR_RESET "\n",
" ");
```

```
        printf("%30s %s\n", " ", "------------");
        printST();

        printf("\n\n%30s" ANSI_COLOR_CYAN "CONSTANT TABLE" ANSI_COLOR_RESET
"\n", " ");
        printf("%30s %s\n", " ", "--------------");
        printCT();
    }
}

void yyerror(char *s)
{
    printf("%d %s %s\n", yylineno, s, yytext);
    flag=1;
    printf(ANSI_COLOR_RED "Status: Parsing Failed - Invalid\n"
ANSI_COLOR_RESET);
}

void ins()
{
    insertSTtype(curid,curtype);
}

void insV()
```

```
{
   insertSTvalue(curid,curval);
}


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

Explanation:

The lex code is detecting the tokens from the source code and returning the corresponding token to the parser. In phase 1, we were printing the token, and now we are returning the token so that parser uses it for further computation. We are using the symbol table and the constant table of the previous phase only. We added functions like insertSTtype(), insertSTvalue() and insertSTline() to the existing functions. Lexical Analyser installs the token in the symbol table, whereas parser calls these functions to add the value of attributes like data type, the value assigned to the identifier, and where the identifier was declared i.e., updates the information in the symbol table.

Declaration Section

In this section we have included all the necessary header files,function declaration and flag that was needed in the code. Between declaration and rules section we have listed all the tokens which are returned by the lexer according to the precedence order. We also declared the operators here according to their associativity and precedence.This ensures the grammar we are giving to the parser is unambiguous as LALR(1) parser cannot work with ambiguous grammar.

Rules Section

In this section production rules for the entire C language are written. The rules are written in such a way that there is no left recursion and the grammar is also deterministic. Non-deterministic grammar was converted to deterministic by applying left factoring. This was done so that grammar is for LL(1) parser. This is so because all LL(1) grammar are LALR(1) according to the concepts. The grammar productions does the syntax analysis of the source code. When a complete statement with proper syntax is matched by the parser.

C-Program Section

In this section the parser links the extern functions,variables declared in the lexer, external files generated by the lexer etc. The main function takes the input source code file and prints the final symbol table.

Test Cases

Without Errors:

Test case 1 (without error)

```
//modifiers

//arithmetic operation

//logical operations


#include<stdio.h>

int main()

{

   long int a, b;

   unsigned long int x;

   signed short int y;

   signed short z;

   int w;

   a = 23;

   b = 15;

   int c = a + b;

   printf("%d",c);

   c = a - b;

   printf("%d",c);

   c = a * b;
```

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

   c = a/b;

   printf("%d",c);

   c = a%b;

   printf("%d",c);


   c = (a>=b);

   printf("%d",c);

   c = (a<=b);

   printf("%d",c);

   c = (a==b);

   printf("%d",c);

   c = (a!=b);

   printf("%d",c);

}
```

Test case 2 (without error)

```
#include<stdio.h>

#define NUM 5


int main()

{

char A[] = "#define MAX 10";
```

```
char B[ ] = "Hello";

char ch  = 'B';

unsigned a = 1;

printf("String = %s Value of Pi = %f", A, 3.14);


   return 0;

}
```

Test Case 3 (without error)

```
#include<stdio.h>


int square(int a)

{

   return(a*a);

}


struct abc

{

  int a;

  char b;

};
```

```
int main()

{

   struct abc A;

   A.a = 2;

   int num = 2;

   int num2 = square(num);


   printf("Square of %d is %d", num, num2);


   return 0;

}
```

Test Case 4 (without error)

```
#include<stdio.h>


int main()

{

   int a = 2;

   printf("%d",a);

   a++;

   int b = 4;

   int c = 3;
```

```
   int b = 8;
   int c = 3;
   int d = c*(a+b);
   a--;
}
```

Test case 5 (without error)

```
// Mixture of tests, comments


#include<stdio.h>
int main()
{
   int a, b;
   char c;
   for(a = 0; a < 29; a++)
   {
       if(a < 15) {
           printf("Hello World");
       }
   }
   int x = a + b;
```

```
   // Single Line Comment
   /* This is a
      multi-line comment */

   int var1;
   char var2;
   printf("%d",x);
}
```

Test case 6 (without error)

```
#include<stdio.h>

int fun(char x){
   return x*x;
}

void main(){
   int a=2,b,c,d,e,f,g,h;

   c=a+b;
   d=a*b;
   e=a/b;
   f=a%b;
```

```
   g=a&&b;

   h=a||b;

   h=a*(a+b);

   h=a*a+b*b;

   h=fun(b);


   //This Test case contains operator,structure,delimeters,Function;

}
```

With Errors:

Test case 1 (with error)

```
#include<stdio.h>

int main()

{

   int a,b;

   int c=a+b+;

   return 0;

}
```

Test case 2 (with error)

```
#include stdio.h>
```

Test case 3 (with error)

```
#include <stdio.h>

int main()
```

```
{

   int 9abi = 10;

}
```

Test Case 4 (with error)

```
// Implicit Error that our Language doesn't support


#include<stdio.h>


int main() {

   char @hello;

   @hello = 'c';

}
```

Test case 5 (with error)

```
#include<stdio.h>


struct student

{

 int rollNum;

 int marks;

};


int main()

{
```

```
int a = 1, b=0;

struct student student1;

student1.rollNum = 1;

student1.marks = 90;



if(a >= 1 && a <= 10)

   b++;



else

      {  b--;

       /* }

}
```

Implementation:

The lexer code submitted in the previous phase took care of most of the features of C using regular expressions. Some special corner cases were taken care of using custom regex. These were:

A. The Regex for Identifiers

B. Multiline comments should be supported

C. Literals

D. Error Handling for Incomplete String

E. Error Handling for Nested Comments

The parser code requires exhaustive token recognition and because of this reason, we utilized the lexer code given under the C specifications with the parser. The parser implements C grammar using a number of production rules. The parser takes tokens from the lexer output, one at a time and applies the corresponding production rules to append to the symbol table with type, value, and line of declaration. If the parsing is not successful, the parser outputs the line number with the corresponding error. The following functions were written in order to maintain the symbol table:

1. LookupST() - This function checks whether the token is already present in the symbol table or not. If yes it returns 1 else 0.(Called by Scanner)

2. InsertST() - This function installs the token in the symbol table if it is not already present along with the token class. (Called by Scanner)

3. InsertSTtype() - This function appends the data type of the identifier in the symbol table. (Called by Parser).

4. InsertSTvalue()- This function appends the value of the identifier in the symbol table. (Called by Parser).

5. InsertSTline()- This function appends the line of the declaration of the identifier in the symbol table. (Called by Parser).

6. LookupCT() - This function checks whether the token is already present in the constant table or not. If yes it returns 1 else 0.(Called by Scanner).

7. InsertST() - This function installs the token in the constant table if it is not already present along with the token class. (Called by Scanner)

8. PrintST() - This function displays the entire content of the symbol table.

9. PrintCT() - This function displays the entire content of the constant table.

Result

The Yacc Script was able to successfully parse all the tokens generated the flex script for C. The type, value, and line of the declaration was returned as an output for all the identifiers and constants present in the program. To handle the error messages the line number was returned along with the syntax error message. Thus the following Yacc script is able to parse the tokens and generate error messages for the C program.

Valid Test Cases:

Test Case 1: modifiers, arithmetic operation, and logical operations.

Output:

```
================================  Running TestCase 10  ================================
Status: Parsing Complete - Valid
                          SYMBOL TABLE
                           ------------
   SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
--------------------------------------------------------------------------------
        a |      Identifier |        int |         23 |          8
        b |      Identifier |        int |         15 |          8
        c |      Identifier |        int |            |         15
        w |      Identifier |        int |            |         12
        x |      Identifier |        int |            |          9
        y |      Identifier |        int |            |         10
        z |      Identifier |      short |            |         11
   signed |         Keyword |            |            |         10
     main |      Identifier |        int |            |          6
    short |         Keyword |            |            |         10
      int |         Keyword |            |            |          6
 unsigned |         Keyword |            |            |          9
   printf |      Identifier |            |       "%d" |         16
     long |         Keyword |            |            |          8


                          CONSTANT TABLE
                           -------------
     NAME |            TYPE
--------------------------------------------------------------------------------
     "%d" | String Constant
       15 | Number Constant
       23 | Number Constant
```

Status: PASS

Test Case 2: array and print statements.

Output:

```
================================  Running TestCase 8  ================================
Status: Parsing Complete - Valid
                            SYMBOL TABLE
                             ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------------
         A | Array Identifier |       char |          0 |          6
         B | Array Identifier |       char |    "Hello" |          7
      char |         Keyword |            |            |          6
         a |      Identifier |       char |          1 |          9
      main |      Identifier |        int |            |          4
        ch |      Identifier |       char |        'B' |          8
       int |         Keyword |            |            |          4
  unsigned |         Keyword |            |            |          9
    printf |      Identifier |            | "String = %s Value of Pi = %f" |         10
    return |         Keyword |            |            |         12


                            CONSTANT TABLE
                             --------------
      NAME |            TYPE
-------------------------------------------------------------------------------------
      3.14 | Floating Constant
"String = %s Value of Pi = %f" | String Constant
       'B' | Character Constant
"#define MAX 10" | String Constant
   "Hello" | String Constant
         0 | Number Constant
         1 | Number Constant
```

Test Case 3: calculate square of a number

Output:

```
================================  Running TestCase 7  ================================
Status: Parsing Complete - Valid
                            SYMBOL TABLE
                             ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------------
         A |      Identifier |     struct |            |         16
      char |         Keyword |            |            |         11
       num |      Identifier |        int |          2 |         18
         a |      Identifier |        int |          2 |          3
         b |      Identifier |       char |            |         11
    square |      Identifier |        int |            |          3
      num2 |      Identifier |        int |          0 |         19
      main |      Identifier |        int |            |         14
       int |         Keyword |            |            |          3
       abc |      Identifier |     struct |            |          8
    printf |      Identifier |            | "Square of %d is %d" |         21
    return |         Keyword |            |            |          5
    struct |         Keyword |            |            |          8


                            CONSTANT TABLE
                             --------------
      NAME |            TYPE
-------------------------------------------------------------------------------------
"Square of %d is %d" | String Constant
         0 | Number Constant
         2 | Number Constant
```

Test Case 4: arithmetic operators

Output:

```
================================  Running TestCase 6  ================================
Status: Parsing Complete - Valid
                          SYMBOL TABLE
                           ------------
   SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------
        a |      Identifier |        int |          2 |          5
        b |      Identifier |        int |          8 |          8
        c |      Identifier |        int |          3 |          9
        d |      Identifier |        int |            |         13
     main |      Identifier |        int |            |          3
      int |         Keyword |            |            |          3
   printf |      Identifier |            |       "%d" |          6


                          CONSTANT TABLE
                           --------------
     NAME |            TYPE
-------------------------------------------------------------------------------
      "%d" | String Constant
        2 | Number Constant
        3 | Number Constant
        4 | Number Constant
        8 | Number Constant
```

Test Case 5: other tests and comments

Output:

```
================================  Running TestCase 13  ================================
Status: Parsing Complete - Valid
                          SYMBOL TABLE
                           ------------
   SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------
     char |         Keyword |            |            |          7
        a |      Identifier |        int |         15 |          6
        b |      Identifier |        int |            |          6
        c |      Identifier |       char |            |          7
        x |      Identifier |        int |            |         14
      for |         Keyword |            |            |          8
     var1 |      Identifier |        int |            |         18
     var2 |      Identifier |       char |            |         19
     main |      Identifier |        int |            |          4
       if |         Keyword |            |            |         10
      int |         Keyword |            |            |          4
   printf |      Identifier |            |       "%d" |         11


                          CONSTANT TABLE
                           --------------
     NAME |            TYPE
-------------------------------------------------------------------------------
"Hello World" | String Constant
      "%d" | String Constant
       15 | Number Constant
       29 | Number Constant
        0 | Number Constant
```

Test Case 6: operators, delimiters and functions

Output:

```
================================  Running TestCase 2  ================================
Status: Parsing Complete - Valid
                          SYMBOL TABLE
                           ------------
   SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
--------------------------------------------------------------------------------
     char |         Keyword |            |            |          3
        a |      Identifier |        int |          2 |          8
        b |      Identifier |        int |            |          8
        c |      Identifier |        int |            |          8
        d |      Identifier |        int |            |          8
        e |      Identifier |        int |            |          8
        f |      Identifier |        int |            |          8
        g |      Identifier |        int |            |          8
        h |      Identifier |        int |            |          8
        x |      Identifier |       char |            |          3
      fun |      Identifier |        int |            |          3
     main |      Identifier |       void |            |          7
      int |         Keyword |            |            |          3
     void |         Keyword |            |            |          7
   return |         Keyword |            |            |          4


                          CONSTANT TABLE
                           --------------
     NAME |            TYPE
--------------------------------------------------------------------------------
        2 | Number Constant
```

Invalid Test Cases:

Test Case 1 Output:

```
==================================  Running TestCase 17  ==================================
5 syntax error ;
Status: Parsing Failed - Invalid
```

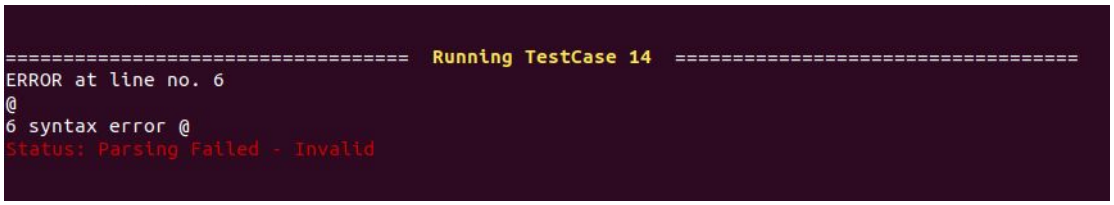
Test Case 2 Output:

```
==================================  Running TestCase 16  ==================================
Error in Pre-Processor directive at line no. 1
#
1 syntax error #
Status: Parsing Failed - Invalid
```

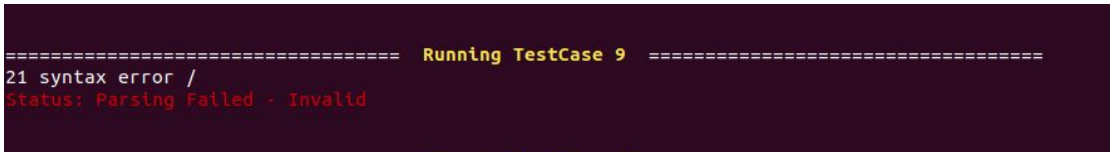
Test Case 3 Output:

```
==================================  Running TestCase 15  ==================================
ERROR at line no. 4
9
4 syntax error 9
Status: Parsing Failed - Invalid
```

Test Case 4 Output:



Test Case 5 Output:



Future Work

The following work contains a YACC script used to analyze various rules of the C language. The YACC script takes care of a number of syntaxes followed by the C language, however, a good portion is still not supported by the script. The future work will be focused on creating a much denser and complete set of syntaxes and rules supported by the C language. The final aim is to provide exhaustive rule support of all the syntaxes supported by the C language in order to handle all the aspects of C language to produce a complete compiler.

References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.
2. The Lex & Yacc Page: http://dinosaur.compilertools.net/
3. John R. Levine, Tony Mason, and Doug Brown. 1992. *Lex & yacc (2nd ed.)*. O'Reilly & Associates, Inc., USA.
4. Lex & Yacc: https://www.epaperpress.com/lexandyacc/
5. Lex & yacc tutorial - Aquamentus: http://aquamentus.com/tut_lexyacc.html