SYNTAX PHASE OR PARSER PHASE

Submitted to:

Ma’am Ujala Saleem

Submitted by:

Tuba Zahid (2K19-BSCS-152)

Najjia Karim (2K19-BSCS-124)

Date of Submission:

12-12-2022

Group Leader:

Ayesha Khan

Project:

MiniC Compiler Design

Parser for the C Language
Compiler Design

This report contains the details of the tasks finished as a part of Phase Two of Compiler Design Lab. We have developed a Parser for C language which makes use of the C lexer to parse the given C input file. In the previous submission, we were using the lexical analyser to generate the stream of tokens from the source code. We used look-ahead for checking errors in comments and some other lexical errors. But lexical analyser cannot detect errors in the structure of a language (syntax), unbalanced parenthesis etc. These errors are handled by a parser.
After the lexical phase, the compiler enters the **syntax analysis phase**. This analysis is done by a parser. The parser uses the stream of tokens from scanner and assigns them datatype if they are identifiers.
The parser code has a functionality of taking input through a file or through standard input. This makes it more user-friendly and efficient at the same time.

Introduction:

Syntactic Analysis and Parser:

After the lexical analysis stage, we get the stream of tokens from source C code which is given as input to the parser. Parser verifies that a string of token names can be generated by the grammar of the source language. We expect the parser to report any syntax errors in an intelligible manner and to recover from the commonly occurring errors to continue processing the remainder of the program. Parser detects the following types of errors:

- Errors in structure
- Missing operator
- Misspelt keywords
- Unbalanced parenthesis

Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. There are generally 2 types of parsers for grammars that we use in class:

1. Top-down
2. Bottom-up

The methods commonly used in compilers can be classified as being either top-down (parse from root to leaves) or bottom-up (parse from leaves to root).

Yacc Script

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine. Lexer can be used to make a simple parser. But it needs making extensive use of the user-defined states.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher-level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.
Yacc is written in portable C. 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; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

```
Definition section
%%
Rules section
%%
C code section
```

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
In the rules section, each grammar rule defines a symbol in terms of:

- Other symbols
- Tokens (or terminal symbols) which come from the lexer.

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.

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs, it is more convenient to place this code in a separate file linked in at compile time.

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 the script using Yacc tool

```
$ yacc -d c_parser.y
```

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[1001];
```

```
struct constanttable
{
char name[100];
char type[100];
int length;
}CT[1001];

int hash(char *str)
{
int value = 0;
for(int i = 0 ; i < strlen(str) ; i++)
{
value = 10*value + (str[i] - 'A');   value = value % 1001;      while(value
< 0)     value = value + 1001;

}
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)%1001)
{
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)%1001)
{
```

```
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)%1001)
{
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 < 1001 ; i++)
{
if(strcmp(ST[i].name,str1)==0)
{
strcpy(ST[i].type,str2);
}
}

}
```

```
void insertSTvalue(char *str1, char *str2)
{
for(int i = 0 ; i < 1001 ; i++)
{
if(strcmp(ST[i].name,str1)==0)
{
strcpy(ST[i].value,str2);
}
}

}
void insertSTline(char *str1, int line)
{
for(int i = 0 ; i < 1001 ; 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)%1001)

{
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 < 1001 ; 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 < 1001 ; 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"          { 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"); insertSTline(yytext,yylineno); 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");
insertSTline(yytext,yylineno); 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_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"



extern char curid[20];
```

```
extern char curtype[20];
extern char curval[20];


%}


%token identifier integer_constant string_constant float_constant character_constant FOR WHILE IF ELSE STRUCT DO
%token VOID INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED INCLUDE DEFINE
BREAK
%token MAIN RETURN


%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(); } identifier_array_type
| expression { ins(); };
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 | STRUCT;
```

```
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(); } '{' V  '}'  ';' ;
structure_declaration
: STRUCT identifier variable_declaration_list;

V
: variable_declaration V ;
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 just 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 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, value assigned to 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 entire C language is 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

```
#include<stdio.h>
/*Test case for operator,nested loops, delimiters, function,
assignments;*/ int fun(int x){     return x*x;
}
void main(){
    int a=2,b,d,e,f,g,h,i,n=10
,x;     d=a*b;     e=a/b;
f=a%b;     g=a&&b;
h=a*a+b*b;     h=fun(b);
    for (i=0;i<n;i++){
        if(i<10){
            int x;
            while(x<10){
x++;
            }
        }
    }
}
```

Test Case 2

```
#include<stdio.h>
//This test case contains nested conditional statement, array and string
declaration int main(){
    char s[10]="Welcome!!";
    char s[]="Welcome!!";
```

```
    int a[2] = {1, 2};
    char S[20];
    if(s[0]=='W'){
        if(s[1]=='e'){
            if(s[2]=='l'){
                printf("Welcome!!");
            }

            else printf("Bug1\n");
        }
        else printf("Bug2\n");
    }
    else printf("Bug3\n"
      ); }
```

Test Case 3:

```
#include<stdio.h>
// This test case contains struct and functions
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); return 0 ;
}
```

With Errors:

Test Case 1:

```
// Invalid syntax missing semicolon.
#include<stdio.h>
int main() {
char hello
    hello = 'c';
}
```

Test Case 2:

```
// Error in structure of assignment
#include
<stdio.h> int
main() {     char
hello;
    'c' = 'x';
}
```

Test Case 3:

```
//Unmatched paranthesis
#include<stdio.h>
int main(){
 int a = 1, b=0;
 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 utilised 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 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 datatype 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 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.

Results:

We were able to successfully parse the tokens recognized by the flex script for C. The output displays the set of identifiers and constants present in the program with their types,values and line of declaration. The parser generates error messages in case of any syntactical errors in the test program.

Valid Test Cases:

Test Case 1: Operator, Nested loops, Delimiters, Function, Assignments Output:

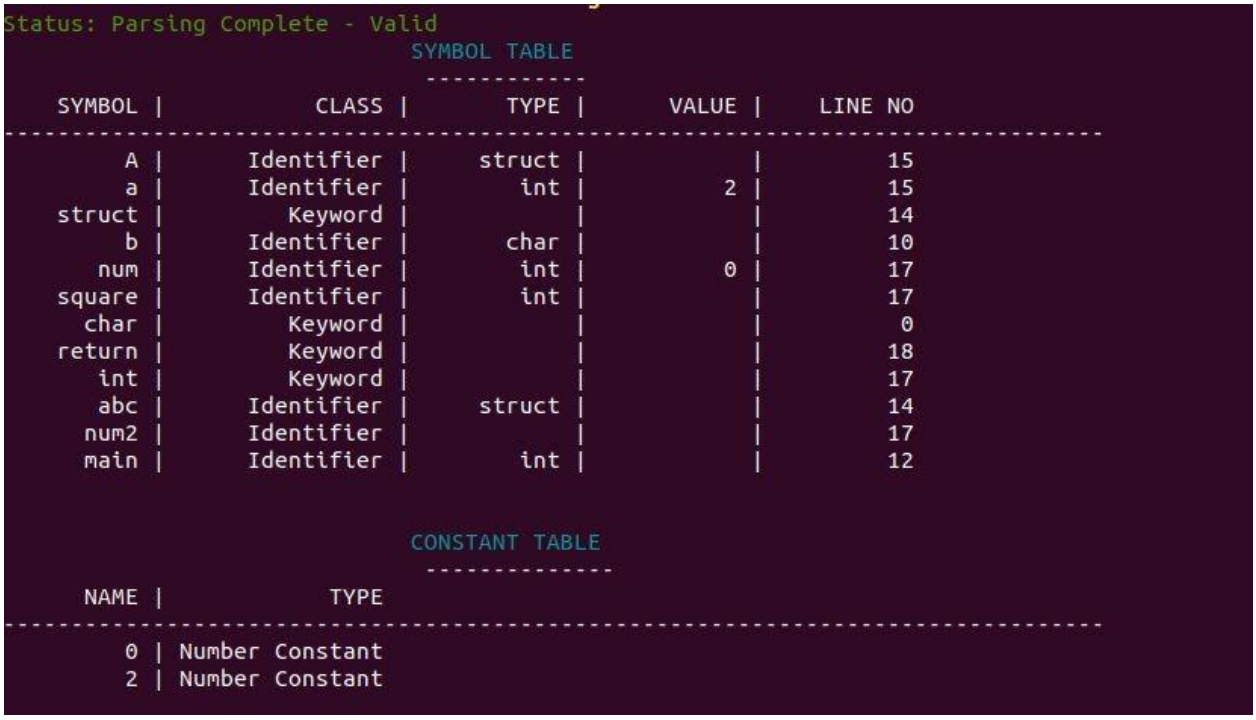
```
Status: Parsing Complete - Valid
                        SYMBOL TABLE
                         ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------
         a |      Identifier |        int |          2 |         16
         b |      Identifier |        int |            |         17
         c |      Identifier |        int |            |          9
         d |      Identifier |        int |            |         10
         e |      Identifier |        int |            |         11
         f |      Identifier |        int |            |         12
         g |      Identifier |        int |            |         13
         h |      Identifier |        int |            |         17
         i |      Identifier |        int |         10 |         19
         n |      Identifier |        int |         10 |         18
         x |      Identifier |        int |         10 |         22
       for |         Keyword |            |            |         18
       fun |      Identifier |        int |            |         17
    return |         Keyword |            |            |          4
        if |         Keyword |            |            |         19
       int |         Keyword |            |            |         20
      main |      Identifier |       void |            |          7
     while |         Keyword |            |            |         21
      void |         Keyword |            |            |          7


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

Fig. 1

Status: Pass

Test Case 2: Structure and Functions

Output:



```
Status: Parsing Complete - Valid
                        SYMBOL TABLE
                         ------------
   SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------
        A |      Identifier |     struct |            |         15
        a |      Identifier |        int |          2 |         15
   struct |         Keyword |            |            |         14
        b |      Identifier |       char |            |         10
      num |      Identifier |        int |          0 |         17
   square |      Identifier |        int |            |         17
     char |         Keyword |            |            |          0
   return |         Keyword |            |            |         18
      int |         Keyword |            |            |         17
      abc |      Identifier |     struct |            |         14
     num2 |      Identifier |            |            |         17
     main |      Identifier |        int |            |         12


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

Fig. 2

Status : PASS

Test Case 3: Convoluted Conditional Statement, Array and String declaration Output:

```
Status: Parsing Complete - Valid
                            SYMBOL TABLE
                             ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------
         S | Array Identifier |       char |            |          7
         a | Array Identifier |        int |            |          6
         s | Array Identifier |       char |        'l' |         10
      char |         Keyword |            |            |          7
        if |         Keyword |            |            |         10
       int |         Keyword |            |            |          6
      main |      Identifier |        int |            |          3
      else |         Keyword |            |            |         18
    printf |      Identifier |            |   "Bug3\n" |         18


                            CONSTANT TABLE
                             --------------
      NAME |            TYPE
-------------------------------------------------------------------------------
  "Bug3\n" | String Constant
  "Bug2\n" | String Constant
  "Bug1\n" | String Constant
       'W' | Character Constant
"Welcome!!" | String Constant
       'e' | Character Constant
       'l' | Character Constant
        10 | Number Constant
        20 | Number Constant
         0 | Number Constant
         1 | Number Constant
         2 | Number Constant
```

Fig. 3.

Status : PASS

Invalid Test Cases

Test Case 1: Invalid syntax missing semicolon

Output:

```
6 syntax error hello
Status: Parsing Failed - Invalid
```

Fig.4.

Status : PASS

Test Case 2: Error in struct of assignment Output:

```
ERROR at line no. 5
'
5 syntax error '
Status: Parsing Failed - Invalid
```

Fig. 5

Status : PASS

Test Case 3: Unmatched parentheses

Output:

```
ERROR at line no. 5
'
5 syntax error '
```

Fig. 6 **Status :**

PASS