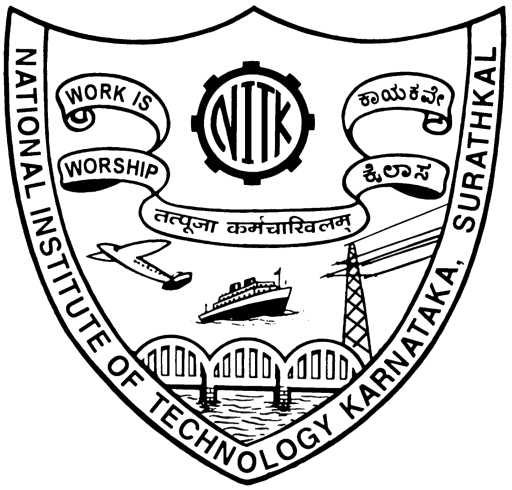
Parser for the C Language



National Institute of Technology Karnataka Surathkal

Date: 7th February 2019

Submitted To:
Prof. P. Santhi Thilagam
CSE Dept, NITK

Group Members:
Namrata Ladda, 16CO121
Mehnaz Yunus, 16CO124
Sharanya Kamath, 16CO140

Abstract:

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.
The parser code has the functionality of taking input through a file or through standard input. This makes it more user-friendly and efficient at the same time.
The parser checks whether the generated tokens form a meaningful expression. This makes use of a context-free grammar that defines algorithmic procedures for components. These work to form an expression and define the particular order in which tokens must be placed. It generates a list of identifiers and functions with their types and also specifies syntax errors if any.

It will have the following features:

- Check the syntax of variable declaration of int, float and char type and also short, long, signed, unsigned subtypes.
- Declaration of arrays with specified datatype (eg: int arr[10])
- Declaration of looping constructs such as while, nested while.
- Proper usage of looping constructs such as while, nested while.
- Declaration and definition of function with one or no argument.
- Check parenthesis balancing
- Precedence and associativity of the operators
- Hashing techniques used to maintain symbol and constant tables.
- Appropriate error messages with line number are displayed.

RESULTS

- Errors in the source program along with appropriate error messages
- Symbol table and constant table will be designed using hashing organization techniques.

TOOLS USED

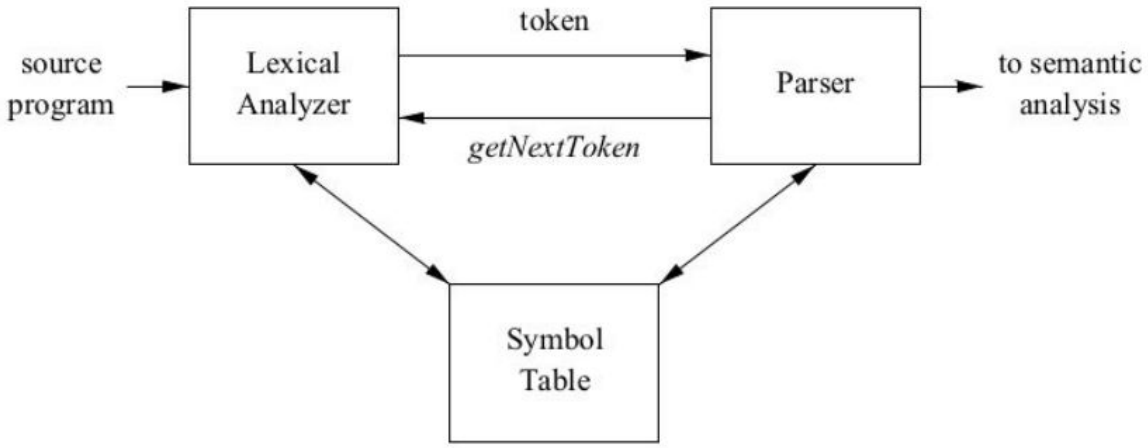
- Yacc
- Flex

Contents :

1. Introduction

1.1 Parser/Syntactic Analysis

In our compiler model, the parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing, as we shall see. Thus, the parser and the rest of the front end could well be implemented by a single module.



Position of parser in compiler model

There are three general types of parsers for grammars: universal, top-down, and bottom-up. As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time.

The most efficient top-down and bottom-up methods work only for sub-classes of grammars, but several of these classes, particularly, LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages. Parsers implemented by hand often use LL grammars. Parsers for the larger class of LR grammars are usually constructed using automated tools.

1.2 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.

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. The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code. 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.

1.3 C Program

The workflow is explained as under:

- Compile the script using Yacc tool
  `$ yacc parser.y`
- Compile the flex *s*cript using Flex tool
- `$ flex lexer.l`
- After compiling the lex file, lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.
- Compilation is done with the options –ll, –ly and -w

```
$ gcc y.tab.c -ll -ly -w -std=c99
```

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

```
$ ./a.out test1.c
```

2. Design of Programs

2.1 Updated Lexer Code:

```
%{
int yylineno;
#include <stdio.h>
%}

%%
\n    { yylineno++; }
"/*"  { multicomment(); }
"//"  { singlecomment(); }
"#include<"([A-Za-z_])*".h>" {}
"#define"([ ])""([A-Za-z_])""([A-Za-z_]|[0-9])*""([ ])""([0-9])+""
                        { }
"#define"([ ])""([A-Za-z_]([A-Za-z_]|[0-9])*)""([
])""(([0-9]+)\.([0-9]+))""          { }
"#define"([ ])""([A-Za-z_]([A-Za-z_]|[0-9])*)""([
])""([A-Za-z_]([A-Za-z_]|[0-9])*)""       { }

\"[^\n]*\"                    { yylval = yytext; return STRING_CONSTANT;
}
\'[A-Za-z_]\'                       { yylval = yytext; return
CHAR_CONSTANT; }
[0-9]+                              { yylval = yytext; return
INT_CONSTANT; }
([0-9]+)\.([0-9]+)                  { yylval = yytext; return
FLOAT_CONSTANT; }
([0-9]+)\.([0-9]+)([eE][-+]?[0-9]+)?      { yylval = yytext; return
FLOAT_CONSTANT; }

"sizeof"                        {  return SIZEOF; }
"++"              {  return INC_OP; }
"--"              {  return DEC_OP; }
"<<"              {  return LEFT_OP; }
">>"              {  return RIGHT_OP; }
"<="              {  return LE_OP; }
">="              {  return GE_OP; }
```

```
"=="             {  return EQ_OP; }
"!="             {  return NE_OP; }
"&&"             {  return AND_OP; }
"||"             {  return OR_OP; }
"*="             {  return MUL_ASSIGN; }
"/="             {  return DIV_ASSIGN; }
"%="             {  return MOD_ASSIGN; }
"+="             {  return ADD_ASSIGN; }
"-="             {  return SUB_ASSIGN; }
"<<="            {  return LEFT_ASSIGN; }
">>="            {  return RIGHT_ASSIGN; }
"&="             {  return AND_ASSIGN; }
"^="             {  return XOR_ASSIGN; }
"|="             {  return OR_ASSIGN; }
"char"              {  yylval = yytext; return CHAR; }
"short"             {  yylval = yytext; return SHORT; }
"int"            {  yylval = yytext; return INT; }
"long"              {  yylval = yytext; return LONG; }
"signed"         {  yylval = yytext; return SIGNED; }
"unsigned"       {  yylval = yytext; return UNSIGNED; }
"void"              {  yylval = yytext; return VOID; }
"if"             {  return IF; }
"else"              {  return ELSE; }
"while"             {  return WHILE; }
"break"             {  return BREAK; }
"return"         {  return RETURN; }
";"                 {  return(';'); }
("{")            {  return('{'); }
("}")            {  return('}'); }
","                 {  return(','); }
":"                 {  return(':'); }
"="                 {  return('='); }
"("                 {  return('('); }
")"                 {  return(')'); }
("["|"<:")       {  return('['); }
("]"|":>")       {  return(']'); }
"."                 {  return('.'); }
"&"                 {  return('&'); }
"!"                 {  return('!'); }
"~"                 {  return('~'); }
"-"                 {  return('-'); }
"+"                 {  return('+'); }
"*"                 {  return('*'); }
"/"                 {  return('/'); }
"%"                 {  return('%'); }
```

```
"<"                     {  return('<'); }
">"                     {  return('>'); }
"^"                     {  return('^'); }
"|"                     {  return('|'); }
"?"                     {  return('?'); }
[A-Za-z_]([A-Za-z_]|[0-9])*              {  yylval = yytext; return
IDENTIFIER;  }
[ \t\v\n\f]       {  }
.                 { }
%%
yywrap()
{
      return(1);
}

multicomment()
{
      char c, c1;
      while ((c = input()) != '*' && c != 0);
      c1=input();
      if(c=='*' && c1=='/')
      {
            c=0;
      }
      if (c != 0)
            putchar(c1);
}
singlecomment()
{
      char c;
      while(c=input()!='\n');
      if(c=='\n')
            c=0;
      if(c!=0)
            putchar(c);
}
```

2.2 Parser Code:

```
%nonassoc NO_ELSE
%nonassoc ELSE
%left '<' '>' '=' GE_OP LE_OP EQ_OP NE_OP
%left  '+'  '-'
%left  '*'  '/' '%'
%left  '|'
```

```
%left  '&'
%token IDENTIFIER STRING_CONSTANT CHAR_CONSTANT INT_CONSTANT FLOAT_CONSTANT
SIZEOF
%token INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME DEF
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOID
%token IF ELSE WHILE CONTINUE BREAK RETURN
%start start_state
%nonassoc UNARY
%glr-parser

%{
#include<string.h>
char type[100];
char temp[100];
%}

%%

start_state
      : global_declaration
      | start_state global_declaration
      ;

global_declaration
      : function_definition
      | variable_declaration
      ;


function_definition
      : declaration_specifiers declarator compound_statement
      | declarator compound_statement
      ;

fundamental_exp
      : IDENTIFIER
      | STRING_CONSTANT       { constantInsert($1, "string"); }
      | CHAR_CONSTANT     { constantInsert($1, "char"); }
      | FLOAT_CONSTANT    { constantInsert($1, "float"); }
      | INT_CONSTANT                { constantInsert($1, "int"); }
      | '(' expression ')'
      ;
```

```
secondary_exp
      : fundamental_exp
      | secondary_exp '[' expression ']'
      | secondary_exp '(' ')'
      | secondary_exp '(' arg_list ')'
      | secondary_exp '.' IDENTIFIER
      | secondary_exp INC_OP
      | secondary_exp DEC_OP
      ;

arg_list
      : assignment_expression
      | arg_list ',' assignment_expression
      ;

unary_expression
      : secondary_exp
      | INC_OP unary_expression
      | DEC_OP unary_expression
      | unary_operator typecast_exp
      ;

unary_operator
      : '&'
      | '*'
      | '+'
      | '-'
      | '~'
      | '!'
      ;

typecast_exp
      : unary_expression
      | '(' type_name ')' typecast_exp
      ;

multdivmod_exp
      : typecast_exp
      | multdivmod_exp '*' typecast_exp
      | multdivmod_exp '/' typecast_exp
      | multdivmod_exp '%' typecast_exp
      ;
```

```
addsub_exp
      : multdivmod_exp
      | addsub_exp '+' multdivmod_exp
      | addsub_exp '-' multdivmod_exp
      ;

shift_exp
      : addsub_exp
      | shift_exp LEFT_OP addsub_exp
      | shift_exp RIGHT_OP addsub_exp
      ;

relational_expression
      : shift_exp
      | relational_expression '<' shift_exp
      | relational_expression '>' shift_exp
      | relational_expression LE_OP shift_exp
      | relational_expression GE_OP shift_exp
      ;

equality_expression
      : relational_expression
      | equality_expression EQ_OP relational_expression
      | equality_expression NE_OP relational_expression
      ;

and_expression
      : equality_expression
      | and_expression '&' equality_expression
      ;

exor_expression
      : and_expression
      | exor_expression '^' and_expression
      ;

unary_or_expression
      : exor_expression
      | unary_or_expression '|' exor_expression
      ;

logical_and_expression
      : unary_or_expression
      | logical_and_expression AND_OP unary_or_expression
      ;
```

```
logical_or_expression
	: logical_and_expression
	| logical_or_expression OR_OP logical_and_expression
	;

conditional_expression
	: logical_or_expression
	| logical_or_expression '?' expression ':' conditional_expression
	;

assignment_expression
	: conditional_expression
	| unary_expression assignment_operator assignment_expression
	;

assignment_operator
	: '='
	| MUL_ASSIGN
	| DIV_ASSIGN
	| MOD_ASSIGN
	| ADD_ASSIGN
	| SUB_ASSIGN
	| LEFT_ASSIGN
	| RIGHT_ASSIGN
	| AND_ASSIGN
	| XOR_ASSIGN
	| OR_ASSIGN
	;

expression
	: assignment_expression
	| expression ',' assignment_expression
	;

constant_expression
	: conditional_expression
	;

variable_declaration
	: declaration_specifiers init_declarator_list ';'
	| error
	;
```

```
declaration_specifiers
      : type_specifier  { strcpy(type, $1); }
      | type_specifier declaration_specifiers   { strcpy(temp, $1);
strcat(temp, " "); strcat(temp, type); strcpy(type, temp); }
      ;

init_declarator_list
      : init_declarator
      | init_declarator_list ',' init_declarator
      ;

init_declarator
      : declarator
      | declarator '=' init
      ;

type_specifier
      : VOID                  { $$ = "void"; }
      | CHAR                  { $$ = "char"; }
      | SHORT                 { $$ = "short"; }
      | INT                   { $$ = "int"; }
      | LONG                  { $$ = "long"; }
      | SIGNED          { $$ = "signed"; }
      | UNSIGNED  { $$ = "unsigned"; }
      ;

type_specifier_list
      : type_specifier type_specifier_list
      | type_specifier
      ;

declarator
      : direct_declarator
      ;

direct_declarator
      : IDENTIFIER                                              {
symbolInsert($1, type); }
      | '(' declarator ')'
      | direct_declarator '[' constant_expression ']'
      | direct_declarator '[' ']'
      | direct_declarator '(' parameter_type_list ')'
      | direct_declarator '(' identifier_list ')'
      | direct_declarator '(' ')'
      ;
```

```
parameter_type_list
	: parameter_list
	;

parameter_list
	: parameter_declaration
	| parameter_list ',' parameter_declaration
	;

parameter_declaration
	: declaration_specifiers declarator
	| declaration_specifiers abstract_declarator
	| declaration_specifiers
	;

identifier_list
	: IDENTIFIER
	| identifier_list ',' IDENTIFIER
	;

type_name
	: type_specifier_list
	| type_specifier_list abstract_declarator
	;

abstract_declarator
	: direct_abstract_declarator
	;

direct_abstract_declarator
	: '(' abstract_declarator ')'
	| '[' ']'
	| '[' constant_expression ']'
	| direct_abstract_declarator '[' ']'
	| direct_abstract_declarator '[' constant_expression ']'
	| '(' ')'
	| '(' parameter_type_list ')'
	| direct_abstract_declarator '(' ')'
	| direct_abstract_declarator '(' parameter_type_list ')'
	;

init
	: assignment_expression
```

```
      | '{' init_list '}'
      | '{' init_list ',' '}'
      ;

init_list
      : init
      | init_list ',' init
      ;

statement
      : compound_statement
      | expression_statement
      | selection_statement
      | iteration_statement
      | jump_statement
      ;

compound_statement
      : '{' '}'
      | '{' statement_list '}'
      | '{' declaration_list '}'
      | '{' declaration_list statement_list '}'
      | '{' declaration_list statement_list declaration_list statement_list
'}'
      | '{' declaration_list statement_list declaration_list '}'
      | '{' statement_list declaration_list statement_list '}'
      ;

declaration_list
      : variable_declaration
      | declaration_list variable_declaration
      ;

statement_list
      : statement
      | statement_list statement
      ;

expression_statement
      : ';'
      | expression ';'
      ;

selection_statement
      : IF '(' expression ')' statement %prec NO_ELSE
```

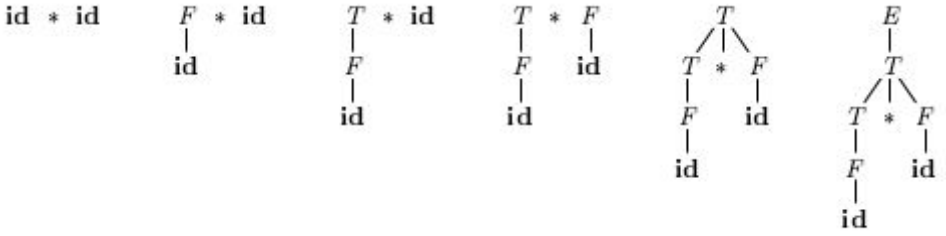
```
      | IF '(' expression ')' statement ELSE statement
      ;

iteration_statement
      : WHILE '(' expression ')' statement
      ;

jump_statement
      : CONTINUE ';'
      | BREAK ';'
      | RETURN ';'
      | RETURN expression ';'
      ;


%%
#include"lex.yy.c"
#include <ctype.h>
#include <stdio.h>
#include <string.h>

struct symbol
{
      char token[100];  // Name of the token
      char dataType[100];           // Date type: int, short int, long int,
char etc
}symbolTable[100000], constantTable[100000];

int i=0; // Number of symbols in the symbol table
int c=0;

//Insert function for symbol table
void symbolInsert(char* tokenName, char* DataType)
{
  strcpy(symbolTable[i].token, tokenName);
  strcpy(symbolTable[i].dataType, DataType);
  i++;
}

void constantInsert(char* tokenName, char* DataType)
{
      for(int j=0; j<c; j++)
      {
            if(strcmp(constantTable[j].token, tokenName)==0)
                  return;
```

```
      }
  strcpy(constantTable[c].token, tokenName);
  strcpy(constantTable[c].dataType, DataType);
  c++;
}

void showSymbolTable()
{


printf("\n\n-----------------------------------------------------------------
------------\n\t\t\t\tSYMBOL TABLE\n
-------------------------------------------------------------------------
\n");
printf("\tSNo\t\t\tLexeme\t\t\tToken\n");
printf("-----------------------------------------------------------------
--------\n");
for(int j=0;j<i;j++)
printf("%10d\t%20s\t\t\t<%s>\t\t\n", j+1, symbolTable[j].token,
symbolTable[j].dataType);
}

void showConstantTable()
{
printf("\n\n-----------------------------------------------------------------
------------\n\t\t\t\tCONSTANT
TABLE\n-----------------------------------------------------------------
-------\n");
printf("\tSNo\t\t\tConstant\t\tDatatype\n");
printf("-----------------------------------------------------------------
--------\n");
for(int j=0;j<c;j++)
printf("%10d\t%20s\t\t\t<%s>\t\t\n",j+1,constantTable[j].token,constantTable
[j].dataType);
}

int err=0;
int main(int argc, char *argv[])
{
      yyin = fopen(argv[1], "r");
      yyparse();
      if(err==0)
            printf("\nParsing complete\n");
      else
            printf("\nParsing failed\n");
```

```
	fclose(yyin);

	showSymbolTable();
	showConstantTable();
	return 0;
}
extern char *yytext;
yyerror(char *s)
{
	err=1;
	printf("\nLine %d : %s\n", (yylineno), s);
	showSymbolTable();
	showConstantTable();
	exit(0);
}
```

Explanation:
The type of parser we have used is **Bottom-Up Parser**. A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Eg: A bottom-up parse for *id * id*



Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. As we shall see, the handle always appears at the top of the stack just before it is identified as the handle.

3. Test Cases:

3.1 Without Error:

Test1.c

```
#include<stdio.h>
int main(){
	int v = 5, c = 6;
	while(v > 0){
		printf("%d",v);
		v--;
```

```
	}
	if(v == 0){
		printf("%d", 2*v + 2);
		v++;
		while(c>0){
			printf("%d is positive\n", c);
			c--;
		}
	}
	else
		printf("negative\n");
}
```

Output for Test1.c

```
$ bash runfile.sh
$ ./a.out Test1.c

Parsing complete


--------------------------------------------------------------------------------
                              SYMBOL TABLE
--------------------------------------------------------------------------------
        SNo                     Lexeme                  Token
--------------------------------------------------------------------------------
         1                      main                    <int>
         2                         v                    <int>
         3                         c                    <int>


--------------------------------------------------------------------------------
                              CONSTANT TABLE
--------------------------------------------------------------------------------
        SNo                     Constant                Datatype
--------------------------------------------------------------------------------
         1                         5                    <int>
         2                         6                    <int>
         3                         0                    <int>
         4                      "%d"                    <string>
         5                         2                    <int>
         6          "%d is positive\n"                  <string>
         7                "negative\n"                  <string>
$
```

Analysis:
There is no error in Test1.c so the parsing completes successfully.

Test2.c

```c
#include<stdio.h>
#define x 3
int main(){
    int a=4;
    if(a<10){
      a=a+1;
      printf("\n%d\n",a);
    }
    else{
      a=a-2;
    }
    return 0;
}
```

Output for Test2.c

```
$ bash runfile.sh
$ ./a.out Test2.c

Parsing complete


---------------------------------------------------------------------------
                              SYMBOL TABLE
---------------------------------------------------------------------------
        SNo                   Lexeme                Token
---------------------------------------------------------------------------
         1                    main                  <int>
         2                       a                  <int>


---------------------------------------------------------------------------
                              CONSTANT TABLE
---------------------------------------------------------------------------
        SNo                   Constant              Datatype
---------------------------------------------------------------------------
         1                       4                  <int>
         2                      10                  <int>
         3                       1                  <int>
         4                  "\n%d\n"                <string>
         5                       2                  <int>
         6                       0                  <int>
$
```

Analysis:

There is no error in Test2.c so the parsing completes successfully.

3.2 With Error

Test3.c

```
#include<stdio.h>
#define x 3

int main()
{
    //int c=4;
    int b=5;
    /* hello world */
    bye*/
    printf("%d",b;
}
```

Output for Test3.c

```
$ bash runfile.sh
$ ./a.out Test3.c

Line 8 : syntax error


--------------------------------------------------------------------------------
                              SYMBOL TABLE
--------------------------------------------------------------------------------
       SNo                    Lexeme                 Token
--------------------------------------------------------------------------------
        1                     main                   <int>
        2                        b                   <int>


--------------------------------------------------------------------------------
                              CONSTANT TABLE
--------------------------------------------------------------------------------
       SNo                    Constant               Datatype
--------------------------------------------------------------------------------
        1                        5                   <int>
$
```

Analysis:
There is a syntax error of unbalanced parenthesis on line 8 in Test3.c so the parsing stops after line 8.

Test4.c

```c
#include<stdio.h>
int main()
{
    int a[4],i=0,b;

    while(i<4){
      a[i]=i;
      i++;
    }
    b = a[0]a[1]+;
    printf("%d",b);
}
```

Output for Test4.c

```
$ bash runfile.sh
$ ./a.out Test4.c

Line 12 : syntax error


--------------------------------------------------------------------------------
                                SYMBOL TABLE
--------------------------------------------------------------------------------
        SNo                     Lexeme                  Token
--------------------------------------------------------------------------------
         1                      main                    <int>
         2                         a                    <int>
         3                         i                    <int>
         4                         b                    <int>


--------------------------------------------------------------------------------
                                CONSTANT TABLE
--------------------------------------------------------------------------------
        SNo                     Constant                Datatype
--------------------------------------------------------------------------------
         1                         4                    <int>
         2                         0                    <int>
$
```

Analysis:
There is a syntax error of missing operator on line 12 in Test4.c so the parsing stops after line 12.

4. 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:

- The Regex for Identifiers
- Multiline comments should be supported
- Literals
- Error Handling for Incomplete String
- 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 generate the symbol table with the variable and function types. If the parsing is not successful, the parser outputs the line number with the corresponding error.

5. Results and Future Work

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.
The parser generates error messages in case of any syntactical errors in the test program.

Future work:

The yacc script presented in this report takes care of all the parsing rules of C language but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle function parameters and other corner cases and thus make it more effective.

References

- http://dinosaur.compilertools.net/yacc/
- https://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/tutorial-large.pdf
- Compilers Principles Techniques and Tools book