Parser for the C Language

National Institute of Technology Karnataka Surathkal

Date:
24/02/2020

Submitted To:
Dr. P. Santhi Thilagam
CSE Dept

Group Members:
17CO227: Niwedita
17CO139: Sanjana P

Abstract:

This report contains the details of the tasks finished as a part of Phase Two of the 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.

Contents:

List of Figures and Tables:

Introduction

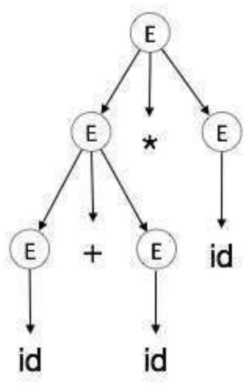
Context Free Grammar

A context-free grammar has four components:
● A set of non-terminals (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
● A set of tokens, known as terminal symbols (Σ). Terminals are the basic symbols from which strings are formed.
● A set of productions (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or non-terminals , called the right side of the production.
● One of the non-terminals is designated as the start symbol (S); from where the production begins. The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Parse Tree:

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree.
Consider the following example with the given Production rules and Input String. Input string : id + id * id
Production rules:
● E →E + E
● E →E * E
● E → id
The left-most derivation is: ● E →E * E
● E →E + E * E
● E → id + E * E
● E → id + id * E
● E → id + id * id

The parse tree for the given input string is :



In a parse tree:
● All leaf nodes are terminals.
● All interior nodes are non-terminals.
● In-order traversal gives original input string.
A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

Parser/Syntactic Analysis

Syntax analysis is a second phase of the compiler design process that comes after lexical analysis. It analyses the syntactical structure of the given input. It checks if the given input is in the correct syntax of the programming language in which the input which has been written. It is known as the Parse Tree or Syntax Tree.

The Parse Tree is developed with the help of pre-defined grammar of the language. The syntax analyzer also checks whether a given program fulfills the rules implied by a context-free grammar. If it satisfies, the parser then creates the parse tree of that source program. Otherwise, it will display error messages.

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.

C Program

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

- Compile the script using Yacc tool
  - $ yacc –d c_parser.y
- Compile the flex script using Flex tool
  - $ flex c_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.
- 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
- 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:

```
alpha        [A-Za-z_]

fl           (f|F|l|L)

ul           (u|U|l|L)*
```

```
digit          [0-9]

space          [ ]

exp            [Ee][+-]?{digit}+


%{

int yylineno;

#include <stdio.h>

#include "y.tab.h"

%}


%%

\n       { yylineno++; }

"/*"     { multicomment(); }

"//"     { singlecomment(); }

"#include<"({alpha})*".h>"

{ }

"#define"({space})""({alpha})""({alpha}|{digit})*""({space})""({digit})+""
{ }

"#define"({space})""({alpha}({alpha}|{digit})*)""({space})""(({digit}+)\.({digit}+))""
{ }

"#define"({space})""({alpha}({alpha}|{digit})*)""({space})""({alpha}({alpha}|{digit})*)""
{ }
```

```
\"[^\n]*\"                                      { yylval = yytext; return
STRING_CONSTANT; }

\'{alpha}\'                                     { yylval = yytext; return
CHAR_CONSTANT; }

{digit}+                                        { yylval = yytext; return
INT_CONSTANT; }

({digit}+)\.({digit}+)                          { yylval = yytext; return
FLOAT_CONSTANT; }

({digit}+)\.({digit}+)([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('?'); }
```

```
{alpha}({alpha}|{digit})*               {  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);

}
```

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
        | 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
          ;

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
        : declaration
        | declaration_list 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 <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------------Symbol Table---------------------\n\nSNo\tToken\t\tDatatype\n\n");

  for(int j=0;j<i;j++)
    printf("%d\t%s\t\t< %s >\t\t\n",j+1,symbolTable[j].token,symbolTable[j].dataType);
}

void showConstantTable()
{
  printf("\n------------Constant Table---------------------\n\nSNo\tConstant\t\tDatatype\n\n");

  for(int j=0;j<c;j++)
    printf("%d\t%s\t\t< %s >\t\t\n",j+1,constantTable[j].token,constantTable[j].dataType);
}

int err=0;
extern FILE *yyin;
extern int yylineno;
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 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 the 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 the 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

Code:

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

```
int main()
{
        int a = 5;

        while(a>0)
        {
                printf("%d",a);
                a--;
                int b = 4;
                while(b>0)
                {
                        printf("%d", a*b);
                        b--;
                }
        }
}
```

Output:

```
Parsing complete

------------Symbol Table---------------------

SNo   Token           Datatype

1     main            < int >
2     a               < int >
3     b               < int >

------------Constant Table---------------------

SNo   Constant                Datatype

1     5               < int >
2     0               < int >
3     "%d"            < string >
4     4               < int >
```

Test Case 2:

Code:

```
#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:

```
Parsing complete

------------Symbol Table--------------------

SNo    Token           Datatype

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 >
```

Test Case 3:

Code:

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

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

Output:

```
Parsing complete

------------Symbol Table--------------------

SNo     Token           Datatype

1       main            < int >
2       b               < int >

------------Constant Table--------------------

SNo     Constant                Datatype

1       5               < int >
2       "%d"            < string >
```

Test Case 4:

Code:

```
//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);
}
```

Output:

```
Parsing complete

------------Symbol Table---------------------

SNo    Token          Datatype

1      main           < int >
2      a              < long int >
3      b              < long int >
4      x              < unsigned long int >
5      y              < signed short int >
6      z              < signed short >
7      w              < int >
8      c              < int >

------------Constant Table---------------------

SNo    Constant              Datatype

1      23             < int >
2      15             < int >
3      "%d"           < string >
```

Test Case 5:

Code:

```
int a;
int b[100];
int c=3;
char d[100] = "kjbk";
char e[] ="jljlj";
int f[10]={0};
int g,h=3,i;
int myfunc()
{

}
```

Output:

Parsing complete

```
------------Symbol Table---------------------

SNo   Token          Datatype

1     a              < int >
2     b              < int >
3     c              < int >
4     d              < char >
5     e              < char >
6     f              < int >
7     g              < int >
8     h              < int >
9     i              < int >
10    myfunc                  < int >

------------Constant Table---------------------

SNo   Constant                Datatype

1     100            < int >
2     3              < int >
3     "kjbk"         < string >
4     "jljlj"        < string >
5     10             < int >
6     0              < int >
```

PASS

With Errors:

Test Case 1:

Code:

```
#include<stdio.h>

int main()
{
    int a = int b = 10;
    int c;

    if(a>b)
    {
      c = 1;
    }
      else
      {
        c = 0;
      }

    printf("Result: %d", c);
}
```

Output:

```
Line 5 : syntax error

------------Symbol Table---------------------

SNo    Token          Datatype

1      main           < int >
2      a              < int >

------------Constant Table---------------------
```

```
SNo    Constant             Datatype
```

Test Case 2:

Code:

```
#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:

```
Line 12 : syntax error

------------Symbol Table---------------------

SNo    Token         Datatype

1      main          < int >
2      a             < int >
3      i             < int >
4      b             < int >

------------Constant Table---------------------

SNo    Constant             Datatype
```

```
1       4               < int >
2       0               < int >
```

Test Case 3:

Code:

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

int main()
{
   int a=4
   while(a<10)
   {
      a=a+1;
      printf("%d\n",a)
   }
}
```

Output:

```
Line 7 : syntax error

------------Symbol Table---------------------

SNo    Token           Datatype

1       main            < int >
2       a               < int >

------------Constant Table---------------------

SNo    Constant                 Datatype

1       4               < int >
```

Test Case 4:

Code:

```
#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--;
        /* }
}
```

Output:

```
Line 3 : syntax error

------------Symbol Table---------------------

SNo    Token           Datatype

1      struct          <  >

------------Constant Table---------------------

SNo    Constant                  Datatype
```

Test Case 5:

Code:

```
//for loop
//continue
//while loop
//do while loop

#include<stdio.h>

int main()
{
   int a=0;
   for (a = 0; a < 10; a++)
      continue;

   while(a>0) {
      a--;
   }

   do {
      a++;
   }while(a<10);
}
```

Output:

```
Line 7 : syntax error

------------Symbol Table---------------------

SNo    Token          Datatype

1      main           < int >
2      a              < int >
```

```
------------Constant Table--------------------

SNo    Constant              Datatype

1      0                < int >
```

PASS

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 reg-ex. 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 utilised the lexer code given under the C specifications with the parser. The parser implements C grammar using a number of production rules.

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.
We have written a function, symlook(), which takes a string (name of the identifier) as input and –
Parser for the C Language :

1. If the identifier is already present in the symbol table, returns the struct symtab entry corresponding to the identifier.
2. If not already present, creates a new struct symtab entry, adds it to the symbol table, and returns the struct symtab.

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. The functions present in the C program are also identified as functions and not as identifiers. The parser generates error messages in case of any syntactical errors in the test program.

Input 1: Sample Programs with most features of C covered

Code:

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

```
int main(){
    int n,i;
    char ch;//Character Datatype

    for (i=0;i<n;i++){
        if(i<10){
            int x;
            while(x<10){
                x++;
            }
        }

    }
    /*
    This File Contains Test cases about Datatypes,Keyword,Identifier,Nested For and
while loop,
    Conditional Statement,Single line Comment,MultiLine Comment etc.*/

}
```

Output:

```
Line 6 : syntax error

------------Symbol Table---------------------

SNo    Token          Datatype

1      main           < int >
2      n              < int >
3      i              < int >
4      ch             < char >

------------Constant Table---------------------

SNo    Constant                Datatype
```

Input 2: Sample C program with convoluted constructions

Code:

```
#include<stdio.h>

int main(){
    char s[10]="Welcome!!";
    char s[]="Welcome!!";
    int a[2] = {1, 2};
    char S[20];
    int p;
    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");

    // int @<-_-= 2;

    //This test case contains nested conditional statement,Array and print statement
    //Also there is an error in declaring integer variable which does not match any regular
expression.
}
```

Output:

```
Parsing complete

------------Symbol Table---------------------

SNo   Token        Datatype
```

```
1       main          < int >
2       s             < char >
3       s             < char >
4       a             < int >
5       S             < char >
6       p             < int >

------------Constant Table--------------------

SNo    Constant               Datatype

1       10                    < int >
2       "Welcome!!"           < string >
3       2                     < int >
4       1                     < int >
5       20                    < int >
6       0                     < int >
7       'W'                   < char >
8       'e'                   < char >
9       'l'                   < char >
10      "Bug1\n"              < string >
11      "Bug2\n"              < string >
12      "Bug3\n"              < string >
```

Input and Output: Invalid C Program

Code:

```
#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--;
        /* }
}
```

Output:

```
Line 3 : syntax error

------------Symbol Table---------------------

SNo    Token          Datatype

1      struct         <  >

------------Constant Table---------------------

SNo    Constant                 Datatype
```

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:

1. http://dinosaur.compilertools.net/
2. http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf

3. Compilers: Principles, Techniques, and Tools: Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman