# Parser for the C Language

**National Institute of Technology Karnataka Surathkal**

**Date:**
16th September 2020

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

**Group Members:**
Niranjan S Yadiyala (181CO136)
Rajath C Aralikatti (181CO241)
Shruthan R (181CO250)
Varun NR (181CO134)

# Abstract

This report contains details of the Parser or the Syntax Analyser for the C language, built as part of Phase 2 of the Compiler Design Project.

**Objective:**
To build a parser for the C language using LEX and YACC scripts.

**Summary:**
The main function of a parser / syntax analyzer is to take in string tokens as input from a C program, compare the tokens with existing grammar rules and thus check the syntactical correctness of the code, while building the corresponding parse tree.

The salient features of the parser include the following :

- Check the syntactical correctness of function declarations and definitions, with or without presence of arguments
- Check the syntax of variable declarations of int, char, float and double types, along with type specifiers like short, long, signed and unsigned.
- Detect declaration of arrays with specified datatype and dimensions.
- Build a symbol table with information regarding symbol names and data types of all the variables and functions declared in the program.
- Detect conditional statements like if, if-else and if-else-if constructs and check their syntactical correctness.
- Detect the proper usage of looping constructs like while loops and nested while loops.
- Check the correctness of arithmetic and logical expressions.
- Check the balancing of different kinds of parentheses.
- Build parse tree for expressions keeping in mind the precedence and associativity of operators.
- Ensures correctness of the general construct of C statements, including presence of commas separating data elements and semicolons at the end of every statement.
- Gives error messages along with the line number in case of presence of any syntactical errors in the code.
- Symbol table and constant table are maintained, and are displayed after the end of the parsing.

**Requirements:**

- GCC, the GNU Compiler Collection
- FLEX (Fast Lexical Analyzer Generator)
- YACC (Yet Another Compiler-Compiler)

# Table of Contents

# 1. Introduction

## 1.1 Parser / Syntax Analysis

When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scanning the input and dividing it into tokens) to target code generation. Syntax Analysis or Parsing is the second phase, which checks the syntactic structure of the given input, i.e. whether the given input is in the correct syntax of the language in which the input has been written or not. It does so by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the predefined grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. If not, error is reported by the syntax analyzer.

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in the figure below, 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. Then the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

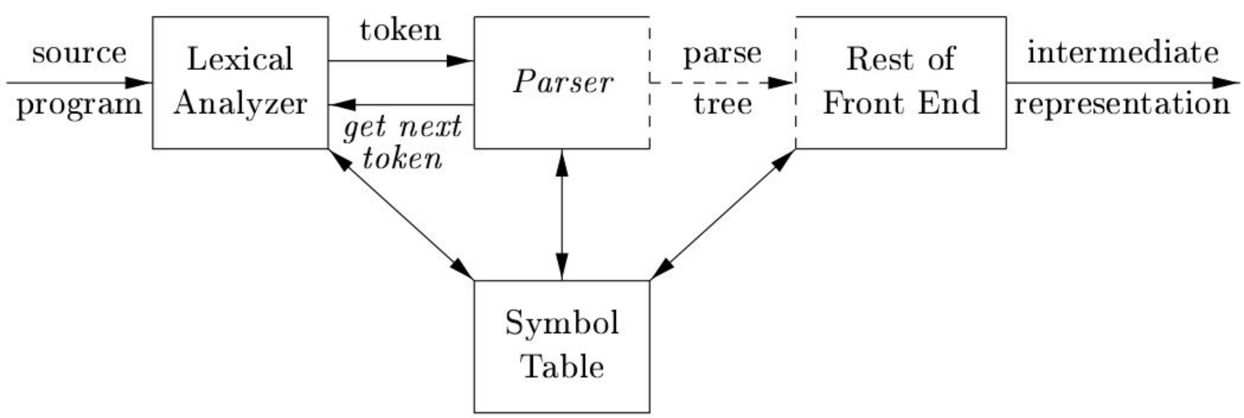



Figure : Position of the parser in the compiler model

Parsing techniques are divided into two different groups:

- Top-Down Parsing: In the top-down parsing construction of the parse tree starts at the root and then proceeds towards the leaves. It searches for a production rule to be used to construct a string.

- Bottom-Up Parsing: In the bottom-up parsing technique the construction of the parse tree starts with the leaf, and then it processes towards its root. it searches for a production rule to be used to reduce a string to get a starting symbol of the grammar. It is also called *shift-reduce parsing*. This type of parsing is created with the help of using some software tools, and this is the kind of parsing used by our compiler.

In both the above methods, 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 Context Free Grammars

A grammar is a set of structural rules which describe a language. Grammars assign structure to any sentence. It is capable of describing many of the syntax of programming languages.

A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple (N, T, P, S) where :

- N is a set of non-terminal symbols.
- T is a set of terminals where $N \cap T$ = NULL.
- P is a set of production rules, P: $N \rightarrow (N \cup T)^*$
- S is the start symbol.

## 1.3 Shift - Reduce Parsing

It is a type of bottom-up parsing with two primary actions, shift and reduce. The input string being parsed consists of two parts :

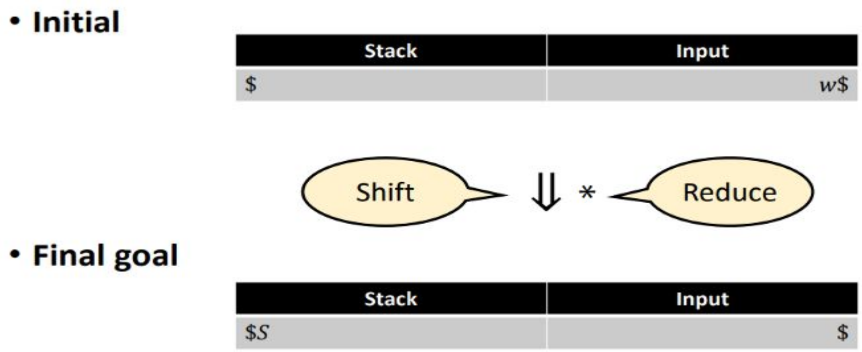
- Left part is a string of terminals and non-terminals, and is stored in a stack
- Right part is a string of terminals read from an input buffer

Shift-Reduce Actions :

- **Shift** is the parser-action of removing the next unread terminal from the input buffer and pushing it into the stack. (The input terminal gets “shifted” to the stack).
- **Reduce** is the parser-action of replacing one or more grammar symbols from the top of the stack that matches a body of a production, with the corresponding production head. The contents on top of the stack which matches the right side of a production is called a handle. The process of replacing a handle with the corresponding production head is called a reduction.
- **Accept** is the parser-action indicating that the entire input has been parsed successfully. The parser executes an accept action only if it reaches the accepting configuration – one in which the input buffer is empty and the stack contains just the start variable followed by '$'.
- **Error** indicates that an error was encountered while parsing the input.

Parsing starts with the input string *w* in the input buffer. Shift and reduce actions are applied until the starting symbol of the grammar is at the top of the stack and the input buffer is empty.

This is the final goal. The input string *w* is successfully parsed if the final goal is reached else parsing is failed.




## 1.4 YACC Script

YACC (Yet Another Compiler Compiler) 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 heart of the yacc specification is the collection of grammar rules. Each rule describes a construct and gives it a name.

YACC is designed for use with C code and generates a parser written in C. The parser is configured for use in conjunction with a lex-generated scanner and relies on standard shared features (token types, yylval, etc.) and calls the function yylex() as a scanner coroutine.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

A YACC program consists of three sections: Declarations, Rules and Auxiliary functions. (Note the similarity with the structure of LEX programs).

```
DECLARATIONS
%%
RULES
%%
AUXILIARY FUNCTIONS
```

**Declarations** : The declarations section consists of two parts: (i) C declarations and (ii) YACC declarations. The C Declarations are delimited by %{ and %}. This part consists of all the declarations required for the C code written in the Actions section and the Auxiliary functions section. YACC copies the contents of this section into the generated y.tab.c file without any modification.

**Rules :** A rule in a YACC program comprises two parts: (i) the production part, having the CFG production rules, and (ii) the action part, which consists of C statements enclosed within '{' and

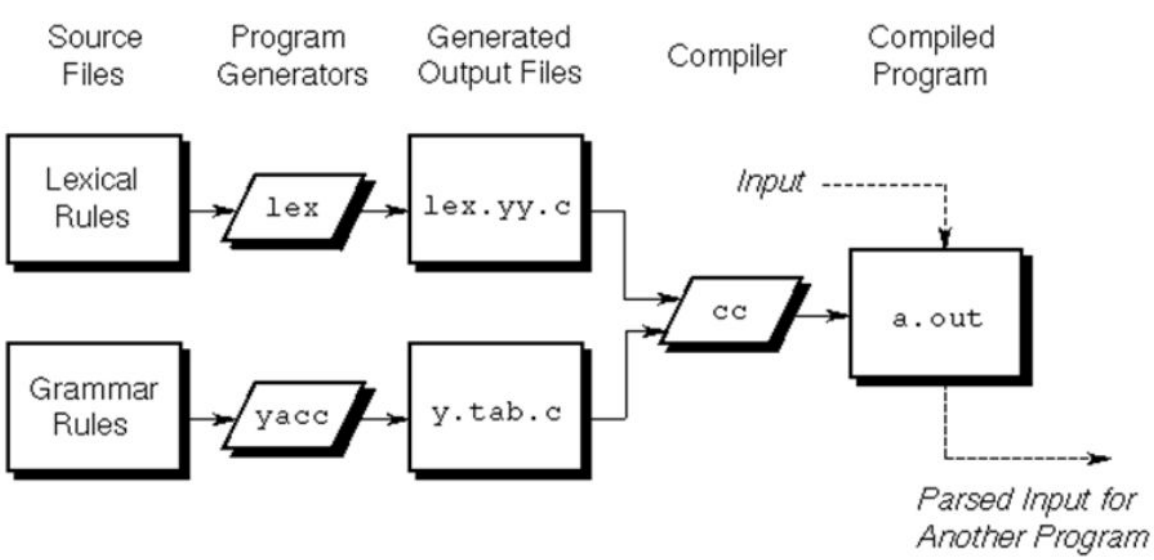
'}', and are executed when the input is matched with the body of a production and a reduction takes place.

**Auxiliary Functions / C code :** The Auxiliary functions section contains the definitions of three mandatory functions main(), yylex() and yyerror(). You may wish to add your own functions (depending on the requirement for the application). Such functions are written in the auxiliary functions section. The main() function must invoke yyparse() to parse the input. The contents here are copied verbatim to the y.tab.c file.

## 1.5 Integration of Lex and YACC

The workflow for integrating the LEX and YACC files, and finally generating the parser is as follows: 1.2 Context Free Grammars

- Compile the flex script using Flex tool:
  - `$ flex lexer.l`
- Compile the parser script using Yacc tool:
  - `$ yacc -d parser.y`
- 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 generated C codes from the lexer and parser are then compiled together :
  - `$ gcc lex.yy.c y.tab.c -w`
- 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

The parser has been implemented in two files:

1. `scanner.l` : The lexical analyzer that scans the input C program, tokenizes the code using various regular expressions and then sends the tokens to the parser.
2. `parser.y` : The syntax analyzer that takes the tokens from the scanner and checks if they form meaningful expressions, with the help of various production rules given by a context-free grammar.

### 2.1 Lexer Code : `scanner.l`

```
%{
   int yylineno;
   #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>
   #include "y.tab.h"
   extern YYSTYPE yylval;
%}

alpha       [A-Za-z_]
fl          (f|F|l|L)
ul          (u|U|l|L)*
digit       [0-9]
space       [ ]
exp         [Ee][+-]?{digit}+

%%
\n      { yylineno++; }
"/*"    { multicomment(); }
"//"    { singlecomment(); }
"#include<"({alpha})*".h>" {}
"#define"({space})""({alpha})""({alpha}|{digit})*""({space})""({digit})+""
{ }
"#define"({space})""({alpha}({alpha}|{digit})*)""({space})""(({digit}+)\.({d
igit}+))""       { }
"#define"({space})""({alpha}({alpha}|{digit})*)""({space})""({alpha}({alpha}
|{digit})*)""    { }
```

```
\"[^\n]*\"  {
   yylval.str = strdup(yytext);
   return STRING_CONSTANT;
}
\'{alpha}\' {
   yylval.str = strdup(yytext);
   return CHAR_CONSTANT;
}
{digit}+  {
   yylval.str = strdup(yytext);
   return INT_CONSTANT;
}
({digit}+)\.({digit}+)  {
   yylval.str = strdup(yytext);
   return FLOAT_CONSTANT;
}
({digit}+)\.({digit}+)([eE][-+]?[0-9]+)?    {
    yylval.str = strdup(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.str = strdup(yytext);
   return CHAR;
}
"short" {
   yylval.str = strdup(yytext);
   return SHORT;
}
"int" {
   yylval.str = strdup(yytext);
   return INT;
}
"long" {
   yylval.str = strdup(yytext);
    return LONG;
}
"signed" {
   yylval.str = strdup(yytext);
   return SIGNED;
}
"unsigned"  {
   yylval.str = strdup(yytext);
   return UNSIGNED;
}
"void" {
   yylval.str = strdup(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.str = strdup(yytext);
   return IDENTIFIER;
}
[ \t\v\n\f]      {  }
<<EOF>>          {
   return EndOfFile;
}
.            { }
%%
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 : `parser.y`

```
%{
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include <ctype.h>

char type[100];
char temp[100];
%}

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

%union {
       char *str;        /* Ptr to constant string (strings are malloc'd) */
   };

%type <str> IDENTIFIER VOID CHAR SHORT INT FLOAT DOUBLE LONG SIGNED UNSIGNED
STRING_CONSTANT CHAR_CONSTANT INT_CONSTANT FLOAT_CONSTANT
%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
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOID
%token IF ELSE WHILE CONTINUE BREAK RETURN
%token EndOfFile

%nonassoc UNARY
%glr-parser
```

```
%%

stmt:   start_state   EndOfFile            {printf("Parsing   successful\n");
showSymbolTable(); showConstantTable(); exit(0);}

start_state
   : global_declaration
   | start_state global_declaration
   ;

global_declaration
   : function_definition
   | declaration
   ;

function_definition
   : declaration_specifiers direct_declarator compound_statement
   | direct_declarator compound_statement
   ;

declaration
   : declaration_specifiers init_declarator_list ';'
   | error
   ;

declaration_specifiers
   : type_specifier
        |   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
   : direct_declarator
   | direct_declarator '=' init
```

```
   ;

init
   : assignment_expression
   | '{' init_list '}'
   | '{' init_list ',' '}'
   ;

init_list
   : init
   | init_list ',' init
   ;

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

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
    ;

statement
    : compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_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 ';'
    ;

expression
    : assignment_expression
    | expression ',' assignment_expression
    ;

assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
```

```
   ;

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

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
   ;

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

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

logical_or_expression
   : logical_and_expression
```

```
   | logical_or_expression OR_OP logical_and_expression
   ;

logical_and_expression
   : unary_or_expression
   | logical_and_expression AND_OP unary_or_expression
   ;

unary_or_expression
   : exor_expression
   | unary_or_expression '|' exor_expression
   ;

exor_expression
   : and_expression
   | exor_expression '^' and_expression
   ;

and_expression
   : equality_expression
   | and_expression '&' equality_expression
   ;

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

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

shift_exp
   : addsub_exp
   | shift_exp LEFT_OP addsub_exp
```

```
   | shift_exp RIGHT_OP addsub_exp
   ;

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

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

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

type_name
   : type_specifier_list
   | type_specifier_list direct_abstract_declarator
   ;

type_specifier_list
   : type_specifier type_specifier_list
   | type_specifier
   ;

type_specifier
   : VOID          { strcpy(type, $1); }
   | CHAR          { strcpy(type, $1); }
   | SHORT         { strcpy(type, $1); }
   | INT           { strcpy(type, $1); }
   | LONG          { strcpy(type, $1); }
   | SIGNED        { strcpy(type, $1); }
   | UNSIGNED      { strcpy(type, $1); }
   ;
```

```
direct_abstract_declarator
   : '(' direct_abstract_declarator ')'
   | '[' ']'
   | '[' constant_expression ']'
   | direct_abstract_declarator '[' ']'
   | direct_abstract_declarator '[' constant_expression ']'
   | '(' ')'
   | '(' parameter_list ')'
   | direct_abstract_declarator '(' ')'
   | direct_abstract_declarator '(' parameter_list ')'
   ;

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

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

constant_expression
   : conditional_expression
   ;
```

```
parameter_list
   : parameter_declaration
   | parameter_list ',' parameter_declaration
   ;

parameter_declaration
   : declaration_specifiers direct_declarator
   | declaration_specifiers direct_abstract_declarator
   | declaration_specifiers
   ;

identifier_list
   : IDENTIFIER
   | identifier_list ',' IDENTIFIER
   ;

%%

extern char *yytext;
extern int yylineno;
extern FILE *yyin;

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

   c++;
}

void showSymbolTable()
{

   printf("\n      Symbol Table         ");
   printf("\n----------------------------\nToken\t\tDatatype\n");
   printf("----------------------------\n");

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

}

void showConstantTable()
{
   printf("\n      Constant Table      ");
   printf("\n----------------------------\nConstant\tDatatype\n");
   printf("----------------------------\n");

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

}

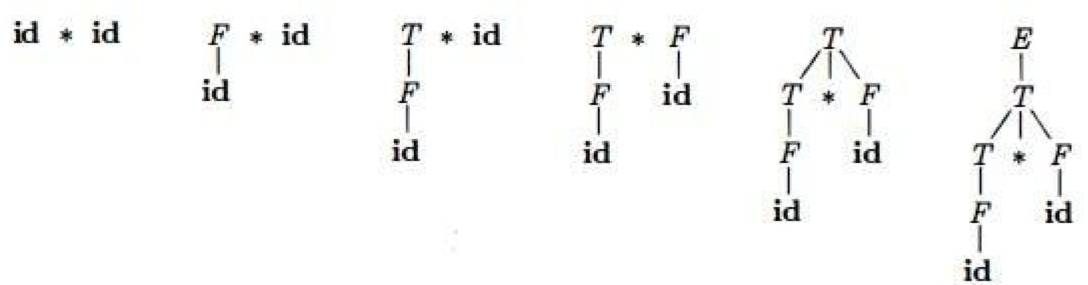
int err=0;
```

```
int main(int argc, char *argv[])
{
   yyin = fopen(argv[1], "r");
   yyparse();
   return 0;
}

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

## 2.3 Explanation

As explained earlier, the YACC tool makes use of the bottom-up parsing technique, which 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).
For example, shown below is the bottom-up parse of the string *id*id* using the unambiguous grammar : E → E + T | T    T → T * F | F    F → id



**Conflicts in Parsing Using YACC :** As noted earlier, YACC uses the shift-reduce parsing methodology. Conflicts arise when the parser is unable to make a decision on the action to execute. These conflicts are practically of two-types: shift/reduce conflict and reduce/reduce conflict.

1. **Shift / Reduce Conflicts:** It occurs when the parser cannot decide whether to shift or to reduce in a configuration where both the actions seem to be viable options.

2. **Reduce / Reduce Conflicts**: It occurs when the parser cannot decide upon which of several possible reductions to make.

The grammar used in our parser does **not** have any shift/reduce or reduce/reduce conflicts. This means that the grammar is totally unambiguous i.e, there exists a unique parse tree for every input string.

## 2.4 First and Follow Sets of Variables

First(A) is a set of terminal symbols that begin in strings derived from the non-terminal A.
Follow(A) is a set of terminal symbols that appear immediately to the right of the non-terminal A.

Given below are the first and follow sets of a few important non-terminal symbols used in our parser:

- ```
  first(start_state)
          = first(global_declaration)
          = first(function_definition) U first(declaration)
          = first(declaration_specifiers) U first(direct_declarator)
          = first(type_specifier) U {IDENTIFIER, ( }
          = {VOID, CHAR, SHORT, INT, LONG, SIGNED, UNSIGNED, IDENTIFIER, ( }
  ```

- `follow(global_declaration) = follow(start_state) = {$}`

- `first(unary_operator) = {&, *, +, -, ~, !}`

- ```
  follow(unary_operator)
         = first(typecast_exp)
         = first(unary_expression) U { ( }
         = first(secondary_exp) U {INC_OP, DEC_OP, ( } U first(unary_operator)
         = first(fundamental_exp) U {INC_OP, DEC_OP, (, &, *, +, -, !, ~}
         = {IDENTIFIER, STRING_CONSTANT, CHAR_CONSTANT,
  FLOAT_CONSTANT, INT_CONSTANT, INC_OP, DEC_OP, (, &, *, +, -, !, ~ }
  ```

- ```
  first(conditional_expression)
         = first(logical_or_expression)
         = first(logical_and_expression)
         = first(unary_or_expression)
         = first(exor_expression)
         = first(and_expression)
         = first(equality_expression)
         = first(relational_expression)
         = first(shift_exp)
         = first(addsub_exp)
         = first(multdivmod_exp)
         = first(typecast_exp)
         = { '(' } U first(unary_expression)
         = { (, INC_OP, DEC_OP, &, *, +, -, ~, !, IDENTIFIER, STRING_CONSTANT,
  CHAR_CONSTANT, FLOAT_CONSTANT, INT_CONSTANT }
  ```
  ```

- first(init)
  = { ‘{’ } U first(assignment_expression)
  = { ‘{’ } U first(conditional_expression) U first(unary_expression)
  = { ‘{’, ‘(’, INC_OP, DEC_OP, &, *, +, -, ~, !, IDENTIFIER, STRING_CONSTANT, CHAR_CONSTANT, FLOAT_CONSTANT, INT_CONSTANT }

- follow(init)
  = follow(init_list) U follow(init_declarator)
  = { ‘}’ , ‘,’ } U follow(init_declarator_list)
  = { ‘}’ , ‘,’ } U { ‘,’ , ‘;’ }
  = { ‘}’ , ‘,’ , ‘;’ }

- first(compound_statement) = { ‘{’ }
- first(selection_statement) = {IF}
- first(iteration_statement) = {WHILE}
- first(jump_statement) = {BREAK, CONTINUE, RETURN}
- first(expression_statement)
  = {‘;’} U first(expression)
  = {‘;’} U first(conditional_expression) U first(unary_expression)
  = {‘;’ , ‘(’, INC_OP, DEC_OP, &, *, +, -, ~, !, IDENTIFIER, STRING_CONSTANT, CHAR_CONSTANT, FLOAT_CONSTANT, INT_CONSTANT}

# 3. Test Cases

## 3.1 Without Errors

If no syntactical errors are found in the program, then the parsing completes, and the symbol table and constant table are displayed.

**Test 1 :**

```
#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 :**

```
niranjan@niranjan-G3-3579:~/Compiler-Design/Parser$ ./run.sh
Parsing successful

      Symbol Table
----------------------------
Token       Datatype
----------------------------
main         int
a            int
b            int
----------------------------


      Constant Table
----------------------------
Constant    Datatype
----------------------------
5            int
0            int
"%d"         string
4            int
----------------------------
```

**Test 2 :**

```
// Has no errors
#include<stdio.h>
#define x 30

int main(void) {
   int m, n;
   m = n = x;
   n = n*10;
   while(m < n) {
       printf("%d", m);
   }
```

```
   return 0;

}
```

**Output:**

```
niranjan@niranjan-G3-3579:~/Compiler-Design/Parser$ ./run.sh
Parsing successful

      Symbol Table
----------------------------
Token       Datatype
----------------------------
main         int
m            int
n            int
----------------------------


      Constant Table
----------------------------
Constant    Datatype
----------------------------
10           int
"%d"         string
0            int
----------------------------
```

**Test 3:**

```
#include<stdio.h>

void print(int x) {
   int i = 0;
   /* hello
   bye*/
   while(i<x){
       printf("%d", x);
   }
}
int main(void) {
  print(10);
}
```

**Output:**

```
niranjan@niranjan-G3-3579:~/Compiler-Design/Parser$ ./run.sh
Parsing successful

      Symbol Table
----------------------------
Token       Datatype
----------------------------
print        void
x            int
i            int
main         int
----------------------------


      Constant Table
----------------------------
Constant    Datatype
----------------------------
0            int
"%d"         string
10           int
----------------------------
```

## 3.2 With Errors

Parsing stops as soon as a syntactical error is encountered in the program, and displays the line number where the error occurred.

**Test 4 :**

```
#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:**

```
niranjan@niranjan-G3-3579:~/Compiler-Design/Parser$ ./run.sh
Parsing Error

Line 5 : syntax error

      Symbol Table
----------------------------
Token            Datatype
----------------------------
main              int
a                 int
----------------------------


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

**Test 5 :**

```
#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:**

```
niranjan@niranjan-G3-3579:~/Compiler-Design/Parser$ ./run.sh
Parsing Error

Line 12 : syntax error

      Symbol Table
----------------------------
Token            Datatype
```

```
---------------------------
main            int
a               int
i               int
b               int
---------------------------

    Constant Table
---------------------------
Constant        Datatype
---------------------------
4               int
0               int
---------------------------
```

**Test 6 :**

```c
#include<stdio.h>

int main() {
   int x = 3
   while(x < 5)
       printf("%d", x);
}
```

**Output:**

```
niranjan@niranjan-G3-3579:~/Compiler-Design/Parser$ ./run.sh
Parsing Error

Line 5 : syntax error

    Symbol Table
---------------------------
Token           Datatype
---------------------------
main            int
x               int
---------------------------


    Constant Table
---------------------------
Constant        Datatype
---------------------------
3               int
---------------------------
```

# 4. Some Implementation Details

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 `y.tab.c` file contains a function `yyparse()` which is an implementation (in C) of a push down automaton. `yyparse()` is responsible for parsing the given input file. The function `yylex()` is invoked by `yyparse()` to read tokens from the input file. Note that the `yyparse()` function is automatically generated by YACC in the `y.tab.c` file. Although YACC declares `yylex()` in the `y.tab.c` file, it does not generate the definition for `yylex()`. Hence the `yylex()` function definition is supplied through the `lex.yy.c` file. Each invocation of `yylex()` must return the next token (from the input stream) to `yyparse()`. The action corresponding to a production is executed by `yyparse()` only after a sufficient number of tokens has been read (through repeated invocations of `yylex()`) to get a complete match with the body of the production.

**Precedence and Associativity of Operators:**
Consider the Context-Free Grammar : $E \rightarrow E + E$, $E \rightarrow E * E$, $E \rightarrow id$. Such a grammar would result in shift-reduce conflicts when run on a YACC script because of its ambiguous nature, i.e, when deriving a string from such a grammar, multiple parse tree structures will be possible and this will lead to conflicts. To disambiguate such grammars, we need to keep in mind the precedence and the associativity of the operators and design the grammar accordingly. An unambiguous grammar of accepting the same language as above would be : $E \rightarrow E + T \mid T$, $T \rightarrow T * F \mid F$, $F \rightarrow id$.

The grammar employed in our parser does not have any conflicts since we have taken into consideration the precedence and associativity of most of the operators supported by C.

- The *precedence* of the operators is as follows (highest precedence on top):
  - `()`
  - `*,/,%`
  - `+,-`
  - `<<,>>`
  - `<, >, <=, >=`
  - `==, !=`
  - &
  - ^
  - |
  - &&
  - ||
  - `?:`
- All binary operators are *left-associative.*

# 5. Results and Future Work

In this second phase of the compiler design project, we have built a parser / syntax analyzer that analyzes the tokens from the source program and checks that they follow the rules of the C syntax perfectly, and gives errors otherwise.
However, syntax analyzers have the following drawbacks -

- It cannot determine if a token is valid.
- It cannot determine if a token is declared before it is being used.
- It cannot determine if a token is initialized before it is being used.
- It cannot determine if an operation performed on a token type is valid or not.

These above tasks are accomplished by the semantic analyzer, which is the next step in building our C compiler. Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not, and thus recognizes semantic errors in a program, like type mismatch, undeclared variable, multiple declaration of a variable in a scope, accessing an out of scope variable etc.

# 6. References


1. A. Aho, M. Lam, R. Sethi and J. Ullman, “Compilers Principles, Tools and Techniques”
2. https://silcnitc.github.io/yacc.html
3. https://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/tutorial-large.pdf
4. https://docs.oracle.com/cd/E19504-01/802-5880/6i9k05dgt/index.html
5. https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm