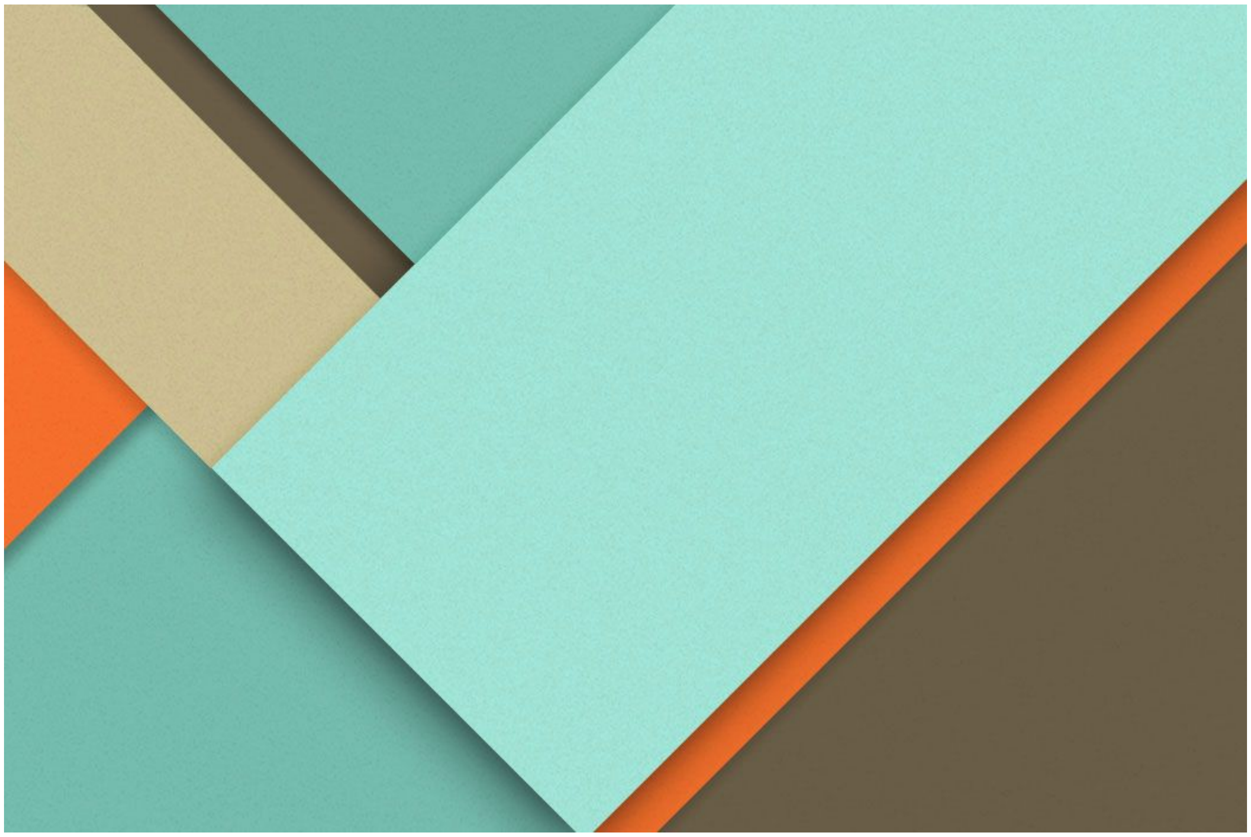


Phase 2 - Syntax Analysis Report

Sagar Bharadwaj 15CO141

Aneesh Aithal 15CO107

Introduction

The second phase of the compiler is syntax analysis or parsing. The parser uses the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. It also checks for correct syntax in the process. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation. The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program.

By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs. In C, for example, a program is made up of functions, a function out of declarations and statements, a statement out of expressions, and so on. The syntax of programming language constructs can be specified by context-free grammars or BNF (Backus-Naur Form) notation. Grammars offer significant benefits for both language designers and compiler writers.

Types of Parsers

The job of the parser is to determine how to represent the input as a derivation of the start state.

There are multiple ways to achieve this:

1. Top-Down Parsing
2. Bottom-Up Parsing

Top Down Parsing

Top-down parsing can be viewed as an attempt to find leftmost derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules. **Example** : LL parser and recursive descent parsers.

Bottom Up Parsing

A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing.

Yacc - Yet Another Compiler Compiler

Yacc (Yet Another Compiler Compiler) is a program that is used to generate a parser for a programming language. It is a LALR (Look Ahead Left-to-Right) parser generator written in a notation similar to BNF. The input to Yacc is a grammar with snippets of C code (called "actions") attached to its rules. Its output is a shift-reduce parser in C that executes the C snippets associated with each rule as soon as the rule is recognized. Typical actions involve the construction of parse trees. Yacc produces only a parser; for full syntactic analysis this requires an external lexical analyzer (here Lex) to perform the first tokenization stage (word analysis), which is then followed by the parsing stage proper.

Code

Lex Code

```
%{

   #define DEBUG 0

   #if defined(DEBUG) && DEBUG > 0
      #define DEBUG_PRINT(fmt, args...) fprintf(stderr, fmt, ##args)
   #else
      #define DEBUG_PRINT(fmt, args...) /* Don't do anything in release builds */
   #endif

   #define RED   "\x1B[31m"
   #define RESET "\x1B[0m"
   #define GRN   "\x1B[32m"
   #define BLU   "\x1B[34m"

   #include "y.tab.h"

   int lineNo = 1;
   int comment = 0;
%}

keyword
char|int|float|short|long|unsigned|signed|main|while|for|break|case|if|else|continue|default|do|return
|void|struct|switch
number         [0-9]
letter         [a-zA-Z]
operator       [+-<>*=/!%^&.]
function       (_|{letter})({letter}|{number}|_)*"()"


%%
\/\/(.)*[\n]                  {lineNo++;}

[/][*]                       { DEBUG_PRINT("%-20s%20s%20d\n", "OPEN COMMENT",yytext, lineNo); comment++; }
[*][/]                       { DEBUG_PRINT("%-20s%20s%20d\n", "CLOSE COMMENT",yytext, lineNo); comment--; }


#                            { if(!comment){
                                  DEBUG_PRINT("%-20s%20s%20d\n","PREPROCESSOR", yytext, lineNo);
                                  return *yytext;
                               }
                            }
```

```
include                    { if(!comment){
                             DEBUG_PRINT("%-20s%20s%20d\n","PREPROCESSOR", yytext, lineNo);
                             return INCLUDE;
                           }
                         }

[\n]                     { lineNo++; }

[{]                      { if(!comment){
                             DEBUG_PRINT("%-20s%20s%20d\n","LEFT BRACE", yytext, lineNo);
                             return *yytext;
                           }
                         }
[}]                      { if(!comment){
                             DEBUG_PRINT("%-20s%20s%20d\n","RIGHT BRACE", yytext, lineNo);
                             return *yytext;
                           }
                         }

\"[^"\n]*["\n]              { if(!comment) {
                           if(yytext[yyleng-1]!='"'){
                             DEBUG_PRINT(RED "Error : Quote unbalanced at line number %d\n" RESET,lineNo);
                             lineNo++;
                           }
                           DEBUG_PRINT("%-20s%20s%20d\n", "STRING", yytext, lineNo);
                           return STRING;
                          }
                         }

int                       { if(!comment){
                           DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                           strcpy(yylval.id, yytext);
                           return INT;
                          }
                         }

float                     { if(!comment){
                           DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                           strcpy(yylval.id, yytext);
                           return FLOAT;
                          }
                         }
char                       { if(!comment){
                           DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                           strcpy(yylval.id, yytext);
                           return CHAR;
                          }
                         }
double                     { if(!comment){
```

```
                        DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                        strcpy(yylval.id, yytext);
                        return DOUBLE;
                      }
                    }
void                  { if(!comment){
                        DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                        strcpy(yylval.id, yytext);
                        return VOID;
                      }
                    }
signed                 { if(!comment){
                        DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                        return SIGNED;
                      }
                    }
unsigned                { if(!comment){
                        DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                        return UNSIGNED;
                      }
                    }
long                  { if(!comment){
                        DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                        return LONG;
                      }
                    }
short                  { if(!comment){
                        DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                        return SHORT;
                      }
                    }
switch                 { if(!comment){
                        DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                        return SWITCH;
                      }
                    }
break                  { if(!comment){
                        DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                        return BREAK;
                      }
                    }
continue                { if(!comment){
                        DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                        return CONTINUE;
                      }
                    }
case                  { if(!comment){
                        DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                        return CASE;
                      }
                    }
default               { if(!comment){
                      DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                      return DEFAULT;
                     }
                    }
for                  { if(!comment){
                      DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                      return FOR;
                     }
                    }
while                 { if(!comment){
                      DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                      return WHILE;
                     }
                    }
do                   { if(!comment){
                      DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                      return DO;
                     }
                    }
if                  { if(!comment){
                      DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                      return IF;
                     }
                    }
else                 { if(!comment){
                      DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                      return ELSE;
                     }
                    }
struct                { if(!comment){
                      DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                      return STRUCT;
                     }
                    }
return                 { if(!comment){
                      DEBUG_PRINT("%-20s%20s%20d\n", "KEYWORD", yytext, lineNo);
                      return RETURN;
                     }
                    }


[(]                  { if(!comment) {
                        DEBUG_PRINT("%-20s%20s%20d\n", "OPEN PARANTHESIS", yytext, lineNo);
                        return OPEN_PAR;
                      }
                    }

[)]                  { if(!comment) {
```

```
                                  DEBUG_PRINT("%-20s%20s%20d\n", "CLOSE PARANTHESIS", yytext, lineNo);
                                  return CLOSE_PAR;
                              }
                          }
[\[]                       { if(!comment) {
                                  DEBUG_PRINT("%-20s%20s%20d\n", "SQUARE BRACKETS", yytext, lineNo);
                                  return *yytext;
                              }
                          }

[\]]                       { if(!comment) {
                                  DEBUG_PRINT("%-20s%20s%20d\n", "SQUARE BRACKETS", yytext, lineNo);
                                  return *yytext;
                              }
                          }

[,]                       { if(!comment) {
                                  DEBUG_PRINT("%-20s%20s%20d\n", "COMMA", yytext, lineNo);
                                  return *yytext;
                              }
                          }


[\t ]                      { ; }

[;]                       { if(!comment) {
                                  DEBUG_PRINT("%-20s%20s%20d\n", "SEMI COLON", yytext, lineNo);
                                  return *yytext;
                              }
                          }

\'.\'                     { if(!comment) {
                              DEBUG_PRINT("%-20s%20s%20d\n", "CHARACTER", yytext, lineNo);
                              return CHARCONST;
                              }
                          }

{number}+(\.{number}+)?e{number}+    { if(!comment) {
                              DEBUG_PRINT("%-20s%20s%20d\n", "FLOAT EXP FORM", yytext, lineNo);
                              return FLOATNUM;
                              }
                          }

{number}+\.{number}+              { if(!comment) {
                              DEBUG_PRINT("%-20s%20s%20d\n", "FLOAT NUMBER", yytext, lineNo);
                              return FLOATNUM;
                              }
                          }
```

```
{number}+                    { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "NUMBER", yytext, lineNo);
                          return NUM;
                          }
                        }

(_|{letter})({letter}|{number}|_)*  { if(!comment){
                          DEBUG_PRINT("%-20s%20s%20d\n", "IDENTIFIER", yytext, lineNo);
                          strcpy(yylval.id, yytext);
                          return ID;
                         }
                        }

"+"                      { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return *yytext;
                          }
                        }
"-"                     { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return *yytext;
                          }
                        }
"*"                      { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return *yytext;
                          }
                        }
"/"                     { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return *yytext;
                          }
                        }

"%"                      { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return *yytext;
                          }
                        }

"^"                      { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return *yytext;
                          }
                        }

"&"                      { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return *yytext;
                          }
                        }
"."                      { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return *yytext;
                          }
                        }
"=="                       { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return EQ;
                          }
                        }
"="                       { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return *yytext;
                          }
                        }
"!="                       { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return NE;
                          }
                        }
"<"                       { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return LT;
                          }
                        }
">"                       { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return GT;
                          }
                        }
"<="                       { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return LE;
                          }
                        }

"+="                       { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return PAS;
                          }
                        }

"-="                       { if(!comment) {
                          DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                          return SAS;
                          }
                        }

"*="                       { if(!comment) {
```

```
                                  DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                                  return MAS;
                                  }
                                }

"/="                            { if(!comment) {
                                  DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                                  return DAS;
                                  }
                                }
">="                              { if(!comment) {
                                  DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                                  return GE;
                                  }
                                }
"&&"                               { if(!comment) {
                                  DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                                  return AND;
                                  }
                                }
"||"                              { if(!comment) {
                                  DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                                  return OR;
                                  }
                                }
"!"                              { if(!comment) {
                                  DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                                  return NOT;
                                  }
                                }

"++"                              { if(!comment) {
                                  DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                                  return PP;
                                  }
                                }
"--"                            { if(!comment) {
                                  DEBUG_PRINT("%-20s%20s%20d\n", "OPERATOR", yytext, lineNo);
                                  return MM;
                                  }
                                }
```

```
{number}({letter}|{number}|_)+      { if(!comment) DEBUG_PRINT(RED "Error : Invalid Token %s at Line %d\n"
RESET, yytext, lineNo); }

.                              { if(!comment) DEBUG_PRINT(RED "Error : Invalid Token %s at Line %d\n" RESET, yytext,
lineNo); }

%%


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

Yacc Code

```
%{
        #include <stdio.h>
        #include <stdlib.h>
        #include "symbolTable.h"

        #define DEBUGY 0
        #if defined(DEBUGY) && DEBUGY > 0
                #define DEBUGY_PRINT(fmt, args...) fprintf(stderr, fmt,
##args)
        #else
                #define DEBUGY_PRINT(fmt, args...)
        #endif

        #define RED   "\x1B[31m"
        #define RESET "\x1B[0m"
        #define GRN   "\x1B[32m"
        #define BLU   "\x1B[34m"

        int yyparse (void);
        int yylex();
        void yyerror(const char * s);
        extern FILE *yyin, *yyout;
        extern char *yytext;
        extern int lineNo;

        char type[100];
%}


%token INT FLOAT CHAR DOUBLE VOID RETURN
```

```
%token SIGNED UNSIGNED LONG SHORT
%token SWITCH BREAK CONTINUE CASE DEFAULT STRUCT
%token FOR WHILE DO
%token IF ELSE
%token NUM FLOATNUM STRING CHARCONST
%token INCLUDE
%token OPEN_PAR CLOSE_PAR

%union {
        char id[100];
}
%token <id> ID
%token <id> INT
%token <id> CHAR
%token <id> FLOAT
%token <id> DOUBLE
%token <id> VOID



%right '=' PAS MAS DAS SAS
%left AND OR NOT PP MM
%left LE GE EQ NE LT GT                         // LE <= GE >= EQ == NE !=
LT < GT >
%left '+' '-' '*' '/' '%' '^' '&' '.'
%start start

%%
start:  FunctionDef
         | Declaration
        | Include
        | FunctionDef start
        | Declaration start
```

```
       | Include start
        ;

IncludeStatement: '#' INCLUDE LT ID GT
                 | '#' INCLUDE LT ID '.' ID GT
                 ;
Include:    IncludeStatement
          ;

FunctionDef: Type ID OPEN_PAR FormalParamList CLOSE_PAR CompoundStatement
{insertSymbolItem($2,"function",lineNo,0);}
            ;
FormalParamList: Type ID
{insertSymbolItem($2,type,lineNo,0);}
               | Type '*' ID
{insertSymbolItem($3,type,lineNo,0);}
               | Type ArrayNotation
{DEBUGY_PRINT("FLIST Call 3\n");}
               | Type ID ',' FormalParamList
{insertSymbolItem($2,type,lineNo,0);}
               | Type '*' ID ',' FormalParamList
{insertSymbolItem($3,type,lineNo,0);}
               | Type ArrayNotation ',' FormalParamList
               |
               ;



Declaration:  Type IDList ';'    {;}
       ;

Type: INT {strcpy(type,$1);}| FLOAT {strcpy(type,$1);}| VOID
{strcpy(type,$1);}| CHAR {strcpy(type,$1);}| DOUBLE {strcpy(type,$1);}|
        Modifiers INT {strcpy(type,$2);}| Modifiers FLOAT
{strcpy(type,$2);}| Modifiers DOUBLE {strcpy(type,$2);}| Modifiers CHAR
{strcpy(type,$2);}

        ;

Modifiers: SHORT | LONG | UNSIGNED | SIGNED

        ;


ArrayNotation: ID '[' ']' {char ar[] = "arr - ";
insertSymbolItem($1,strcat(ar, type),lineNo,0);}

             | ID '[' Expr ']' {char ar[] = "arr - ";
insertSymbolItem($1,strcat(ar, type),lineNo,0);}

             ;


IDList: ArrayNotation

        | ID ',' IDList {insertSymbolItem($1,type,lineNo,0);}

        | '*' ID ',' IDList {insertSymbolItem($2,type,lineNo,0);}

        | ArrayNotation ',' IDList

        | ID {insertSymbolItem($1,type,lineNo,0);}

        | '*' ID {insertSymbolItem($2,type,lineNo,0);}

        | DefineAssign ',' IDList

        | DefineAssign

        ;


DefineAssign: ID '=' Expr

             | ID PAS Expr

             | ID SAS Expr

             | ID MAS Expr

             | ID DAS Expr

             | '*' ID '=' Expr

             | '*' ID PAS Expr

             | '*' ID SAS Expr

             | '*' ID MAS Expr

             | '*' ID DAS Expr
```

```
            | ArrayNotation '=' Expr
            | ArrayNotation PAS Expr
            | ArrayNotation SAS Expr
            | ArrayNotation MAS Expr
            | ArrayNotation DAS Expr
            ;


ParamList: Expr
        | Expr ',' ParamList
        |
        ;

Assignment: ID '=' Expr
            | ID PAS Expr
            | ID SAS Expr
            | ID MAS Expr
            | ID DAS Expr
            | '*' ID '=' Expr
            | '*' ID PAS Expr
            | '*' ID SAS Expr
            | '*' ID MAS Expr
            | '*' ID DAS Expr
            | ArrayNotation '=' Expr
            | ArrayNotation PAS Expr
            | ArrayNotation SAS Expr
            | ArrayNotation MAS Expr
            | ArrayNotation DAS Expr
            | Primary
            ;

Expr: Logical_Expr
```

```
     ;


Logical_Expr: Relational_Expr
             | Logical_Expr AND Relational_Expr
             | Logical_Expr OR Relational_Expr
             | NOT Relational_Expr
             ;

Relational_Expr: Additive_Expr
                | Relational_Expr GT Additive_Expr
                | Relational_Expr LT Additive_Expr
                | Relational_Expr GE Additive_Expr
                | Relational_Expr LE Additive_Expr
                | Relational_Expr EQ Additive_Expr
                | Relational_Expr NE Additive_Expr
                ;


Additive_Expr: Multiplicative_Expr
              | Additive_Expr '+' Multiplicative_Expr
              | Additive_Expr '-' Multiplicative_Expr
              ;
Multiplicative_Expr: Primary
                    | Multiplicative_Expr '*' Primary
                    | Multiplicative_Expr '/' Primary
                    | Multiplicative_Expr '%' Primary
                    ;
Primary: OPEN_PAR Expr CLOSE_PAR
        | NUM | FLOATNUM | CHARCONST | STRING
        | ID                         {DEBUGY_PRINT("Primary
Identifier\n");}
```

```
        | '*' ID                       {DEBUGY_PRINT("Pointer
Identifier\n");}
        | '&' ID                       {DEBUGY_PRINT("Address of
Identifier\n");}
        | '-' Primary
        | '+' Primary
        | ArrayNotation
        | FunctionCall
        | PP ID
        | ID PP
        | MM ID
        | ID MM
        ;

CompoundStatement: '{' StatementList '}'
        ;
StatementList: Statement StatementList
              |
              ;

Statement: WhileStatement
        | Declaration
        | ForStatement
        | IfStatement
       | Assignment    ';'
       | ReturnStatement
       | DoWhileStatement
       | BREAK ';'
       | CONTINUE ';'
        | ';'
       ;
```

```
ReturnStatement: RETURN Expr ';'   {DEBUGY_PRINT("Return Statement
Call\n");}
                ;

WhileStatement: WHILE OPEN_PAR Expr CLOSE_PAR Statement
               | WHILE OPEN_PAR Expr CLOSE_PAR CompoundStatement
               ;

DoWhileStatement: DO CompoundStatement WHILE OPEN_PAR Expr CLOSE_PAR ';'
                 ;



ForStatement: FOR OPEN_PAR Assignment ';' Expr ';' Assignment CLOSE_PAR
Statement
             | FOR OPEN_PAR Assignment ';' Expr ';' Assignment CLOSE_PAR
CompoundStatement
             ;

IfStatement: IF OPEN_PAR Expr CLOSE_PAR Statement ElseStatement
            | IF OPEN_PAR Expr CLOSE_PAR CompoundStatement ElseStatement
            ;

ElseStatement: ELSE CompoundStatement
              | ELSE Statement
              |
              ;

FunctionCall: ID OPEN_PAR ParamList CLOSE_PAR
{DEBUGY_PRINT("Function Call\n");}
               ;

%%
#include<ctype.h>
```

```
int count=0;

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

  if(!yyparse())
                printf("\nParsing complete\n");
        else
                printf(RED "\nParsing failed\n" RESET);


        fclose(yyin);


        showSymbolTable();
   return 0;
}

void yyerror(const char *s) {
        printf(RED "%d : %s %s\n" RESET, lineNo, s, yytext );
}
```

Symbol Table - Header File “symbolTable.h”

```
#ifndef SYMBOLTABLE_H
#define SYMBOLTABLE_H

#include <string.h>
   #define DEBUG 1

   #if defined(DEBUG) && DEBUG > 0
       #define DEBUG_PRINT(fmt, args...) fprintf(stderr, fmt, ##args)
   #else
       #define DEBUG_PRINT(fmt, args...) /* Don't do anything in release 
builds */
   #endif


#define RED   "\x1B[31m"
#define RESET "\x1B[0m"
#define GRN   "\x1B[32m"
#define BLU   "\x1B[34m"

const int symbolTableSize = 1000;
   typedef struct symbolItemStruct{
       char tokenValue[100];
       char tokenType[100];
       int lineNumber;
       struct symbolItemStruct* next;
   } symbolItem;
   symbolItem * symbolTable[1000];
   symbolItem * constantTable[1000];


void initSymbolTable(){
   int i;
   for(i = 0;i<symbolTableSize; i++)
       symbolTable[i] = NULL;

   for(i = 0;i<symbolTableSize; i++)
       constantTable[i] = NULL;
}

int hash(unsigned char *str)
{
   unsigned long hashVar = 5381;
   int c;

   while (c = *str++)
       hashVar = (((hashVar << 5) + hashVar) + c)%1000;

   return hashVar;
}

symbolItem* createSymbolItem(char *tokenValue, char *tokenType, int
lineNumber){
   symbolItem *item = (symbolItem*)malloc(sizeof(symbolItem));
   strcpy(item->tokenValue, tokenValue);
   strcpy(item->tokenType, tokenType);
   item->lineNumber = lineNumber;
   item->next = NULL;
   return item;
}

int lookUpSymbolItem(char * tokenValue){
   int hashIndex = hash(tokenValue);

       symbolItem * temp = symbolTable[hashIndex];
       while(temp!=NULL && strcmp(tokenValue, temp->tokenValue)!=0)
           temp=temp->next;

       if(temp==NULL) return 0;
       else return 1;

}

void insertSymbolItem(char *tokenValue, char *tokenType, int lineNumber,
int tableno){
   if(!lookUpSymbolItem(tokenValue)){
   int hashIndex = hash(tokenValue);

   symbolItem *item = createSymbolItem(tokenValue, tokenType, lineNumber);

   if(tableno == 0)
   {
       symbolItem * temp = symbolTable[hashIndex];
       while(temp!=NULL && temp->next!=NULL)
           temp = temp->next;

       if(temp == NULL)
           symbolTable[hashIndex] = item;
       else
```

```
            temp->next = item;
    }

    else
    {
        symbolItem * temp = constantTable[hashIndex];
        while(temp!=NULL && temp->next!=NULL)
            temp = temp->next;

        if(temp == NULL)
            constantTable[hashIndex] = item;
        else
            temp->next = item;
    }
    }

}

void printSymbolItem(symbolItem * item){
    DEBUG_PRINT("%-20s%10s%20d\n",item->tokenValue, item->tokenType,
item->lineNumber);
}

void showSymbolTable(){
    int i;

DEBUG_PRINT("\n-------------------------------------------------------------
------\n");
    DEBUG_PRINT(BLU "%-20s%10s%24s\n","VALUE","TYPE","LINE NUMBER" RESET);

DEBUG_PRINT("-------------------------------------------------------------
----\n");
```

```
   for(int i=0;i<symbolTableSize;i++){
       symbolItem* temp = symbolTable[i];
       while(temp!=NULL){
           printSymbolItem(temp);
           temp=temp->next;
       }
   }
}

void showConstantTable(){
   int i;

DEBUG_PRINT("\n-----------------------------------------------------------
------\n");
   DEBUG_PRINT(BLU "%-20s%10s%24s\n","VALUE","TYPE","LINE NUMBER" RESET);

DEBUG_PRINT("-------------------------------------------------------------
----\n");

   for(int i=0;i<symbolTableSize;i++){
       symbolItem* temp = constantTable[i];
       while(temp!=NULL){
           printSymbolItem(temp);
           temp=temp->next;
       }
   }
}

#endif
```

Explanation of Implementation

An LALR parser for C language was developed using the YACC tool.

- The Grammar for C language was written in Backus - Naur Form. The production rules were then input into the yacc tool to generate the corresponding C code that could parse a file following those production rules.
- The symbol table routines were shifted to the Parser from the scanner. A new column was added to the symbol table to represent the data type of all identifiers in the symbol table.
- The parser makes a call to the scanner whenever it is in need of a new token. If the token returned is an identifier, the identifier is added to the symbol table along with the respective data type.
- All shift-reduce and reduce-reduce conflicts were taken care of. The grammar used is unambiguous and has no conflicts.
- The YYSTYPE union, which is the data type of yylval variable was defined in parser to be able to hold character arrays. This enabled returning the identifier's token value to parser and thus made updation of symbol table in parser possible.
- The variables' type was identified by the rule corresponding to the variables' declaration.
- Reporting of line number corresponding to a syntax error was made possible by the 'lineNo' variable maintained in the scanner. The same variable was 'extern'ed in parser and was printed whenever a syntax error is encountered.
- The hash organisation of the symbol table was ported from the scanner to the parser.

Test Cases

Test Case - 1

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
// T1
/*
A very Basic Program
Tests for :
- Comment removal (Both Single line and multi line)
- No Errors
*/

int main()
{
   printf("Hello World");

   int a,b,c;
   float z;
   if(a>5 && 5 > 4)
   {
       printf("Hello");
       c = a*5;
   }

   else if(b<3 || a < 3)
   {
       if( x > 5)
       {
           a = 5;
           *b = c*7;
       }

   }
   else
   {
       c =6;
   }

   a = 5;
   b = 6;
   c = 7;
   b = a + c;
   return 0;
}
```

Test Case - 2

```
//T2
/*
Errors include :
- Missing semicolon
- Missing operator
- Invalid function call
*/

#include <stdio.h>
int anyh(int arr[], int l, int r, int x)
{
  if (r >= l)
  {
        int mid;
        mid = l*(r - l)2;    // Missing Operator
        if (arr[mid] == x)
            return mid         // No Semicolon
        if (arr[mid] > x)
            return binarySearch(arr, l, mid-1, x);
        return binarySearch(arr, mid+1, r, x);
  }
  return -1;
}
int main()
{
  int arr[10];
  int n, x, result;
  n =  sizeof(arr)/ sizeof(arr[0]);
  x = 10;
  result = binarySearch(arr, 0 n-1, x);        //No comma - Invalid Function Call
  if(result == -1) printf("Element is not present in array");
  else
   printf("Element is present at index %d", result);
  return 0;
}
```

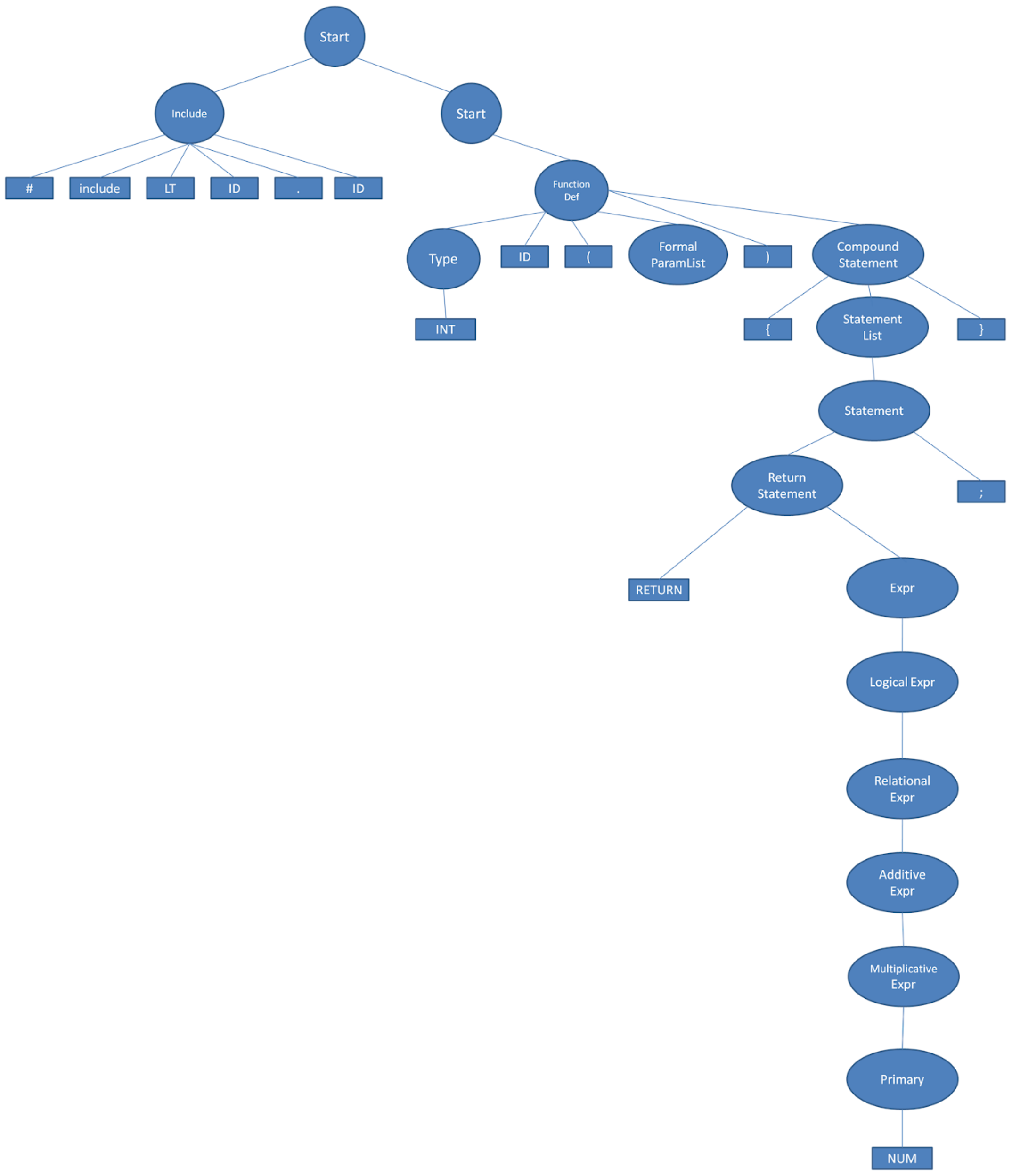
Test Case - 3

```
// T3

/*
   Errors include:
-   Missing paranthesis
-   Missing brace
-   Missing square bracket
-   Invalid operator

*/


#include <stdio.h>
#include <math.h>
void insertionSort(int arr[], int n         //Missing paranthesis
{
  int i, key, j;
  for (i = 1; i < n; i+=1)
  {
      key = arr[i];
      j = i-1;
      while (j >= 0 && arr[j] > key)
                                            //Missing brace
          arr[j+1] = arr[j];
          j = j-1;
      }
      arr[j+1 = key;                        //Missing square bracket
  }
}
void printArray(int arr[], int n)
{
  int i;
  for (i=0; i < n; i++=1)                    //Invalid operator
      printf("%d ", arr[i]);
  printf("\n");
}
int main()
{
   int arr[10],n = 5 + x/2 - z--;
   n = sizeof(arr)/sizeof(arr[0]);
   insertionSort(arr, n);
   printArray(arr, n);
   return 0;
}
```

Test Case - 4

```
// T4
#include <stdio.h>
int main() {
   return 0;
}
```

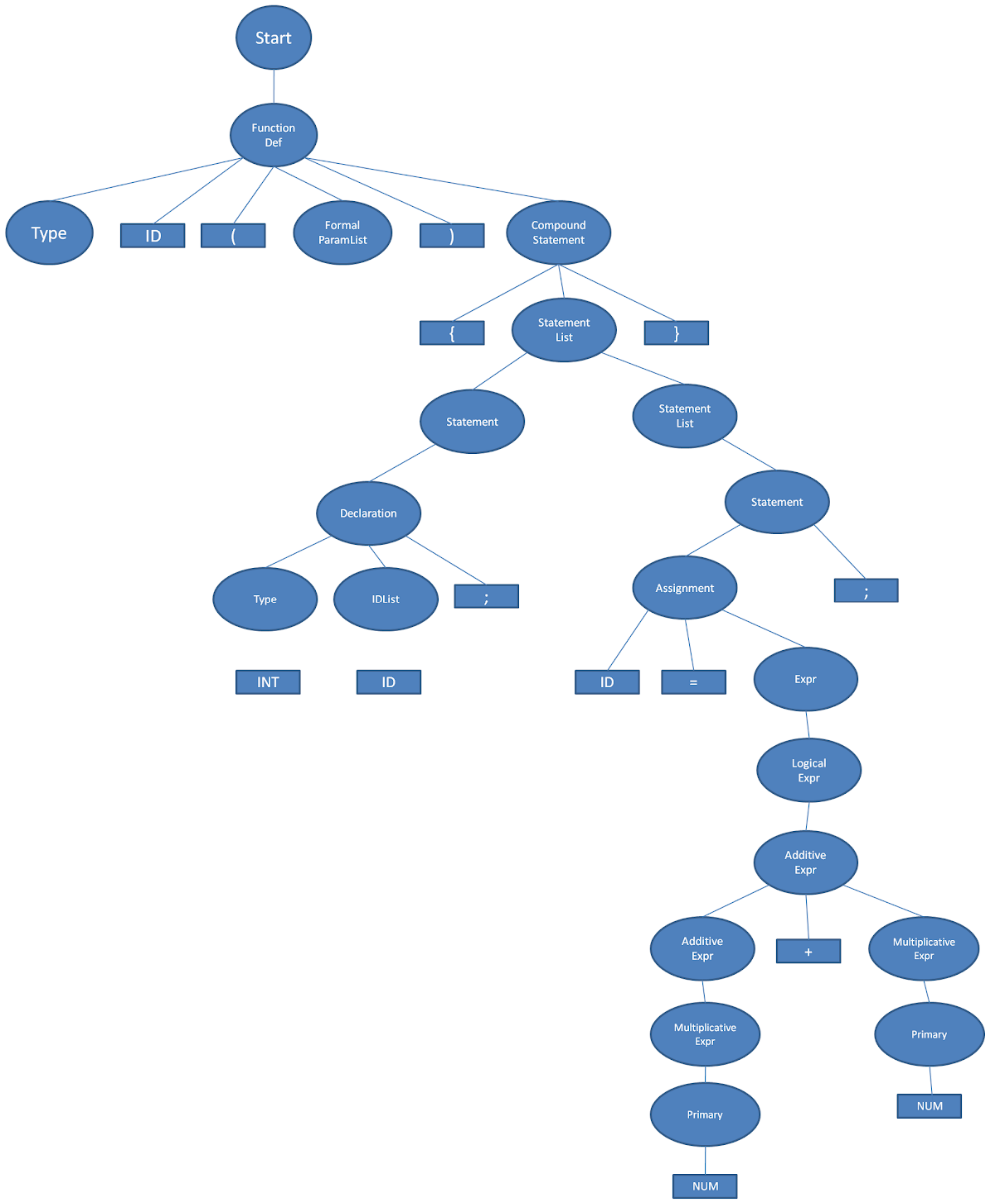
(Zoom in to see the parse Tree. Cicles - Non Terminals; Rectangle - Terminals)



Test Case - 5

```
void main()
{
   int a;
   a = 5 + 4;
}
```

(Zoom in to see the parse Tree. Cicles – Non Terminals; Rectangle – Terminals)



Conclusion

The second phase of the project which involved implementing a parser for C language using YACC tool was successfully completed.

Some of the difficulties faced during this phase of the project included porting symbol table from scanner to parser and eliminating the reduce-reduce and shift-reduce conflicts. Most of the C language constructs were covered in our parser.

The features of our grammar includes

- **Iteration** : For, While, Do While
- **Conditional** : If, If Else, If ElseIf
- Nested form of the above statements
- **Functions**
  - Function Definition
  - Function Declaration
  - Parameter list with expressions support
  - Function call
  - Function call support in expressions
- **Pointers**
  - Referencing and dereferencing operator support in Expressions
- **Arrays**
  - Basic array notation support in :
    - Expressions
    - Declarations
  - Support for expressions in array subscript
- **Data Types**
  - Integer
  - Char

- Modifiers :
  - Long
  - Short
  - Unsigned
  - Signed
- **Operators** : Operator precedence taken care of
  - Arithmetic
  - Relational
  - Logical
  - Referencing and dereference
  - Unary : ++,--,+,-
  - Assignment and Shorthand assignment
- Include Statements
- **Declaration**
  - Multi variable declaration
  - Initialization along with declaration
- Return statement supports recursion