Syntax Analysis Phase Report

National Institute of Technology Karnataka, Surathkal

Submitted to - Ms. Sushmita

Group Members,
Akshay Honrao (15CO218)
A Tarun Karthik (15CO155)

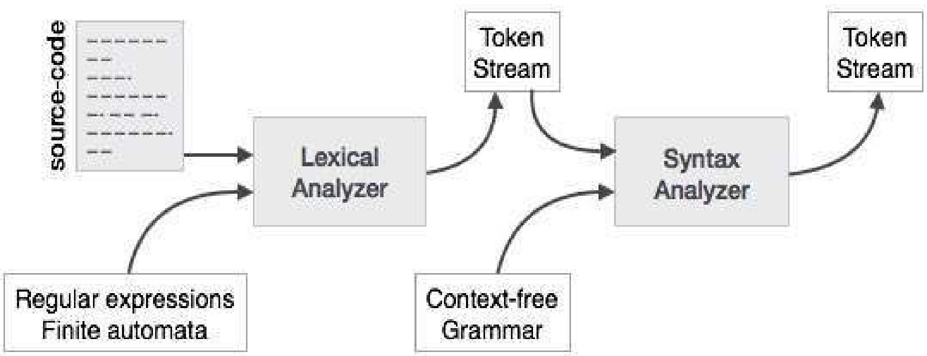
Contents :

Introduction :

Parser :

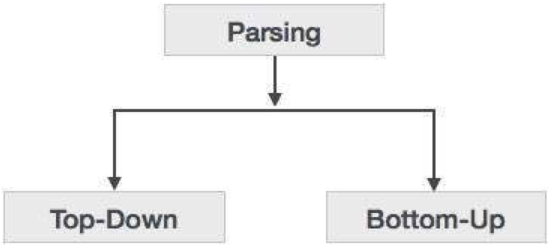
Syntax analysis or parsing is the second phase of a compiler. We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata. CFG, on the other hand, is a superset of Regular Grammar. It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.



This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase. Parsers are expected to parse the whole code even if some errors exist in the program.

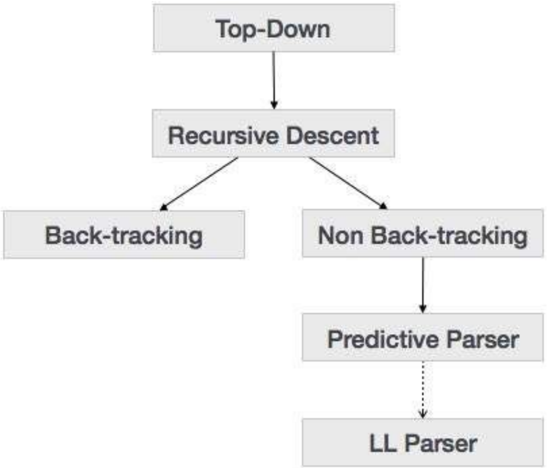
Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types : top-down parsing and bottom-up parsing.



Top-down Parsing :

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

The types of top-down parsing are depicted below -



Recursive Descent Parsing :

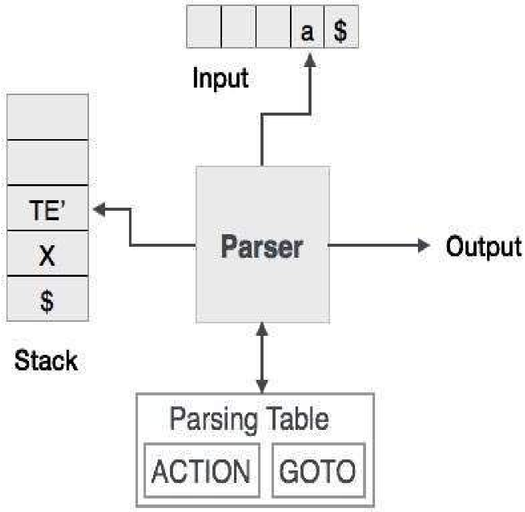
Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any backtracking is known as predictive parsing.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Predictive Parser :

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser backtracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol $to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

LL Parser :

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).

Yacc Script :

A YACC source program is structurally similar to a LEX one.

```
declarations
%%
rules
%%
routines
```

- The declaration section may be empty. Moreover, if the routines section is omitted, the second %% mark may be omitted also.
- Blanks, tabs, and newlines are ignored except that they may not appear in names.

The Declarations Section :

- Declarations of tokens. Yacc requires token names to be declared as such using the keyword %token.
- Declaration of the start symbol using the keyword %start
- C declarations: included files, global variables, types.
- C code between %{ and %}.

Rules Section.

A rule has the form:

```
nonterminal : sentential form
        | sentential form
        .................
        | sentential form
        ;
```

Actions may be associated with rules and are executed when the associated sentential form is matched.

Lex-Yacc Interaction :

yyparse() calls yylex() when it needs a new token.

LEX	YACC
return(TOKEN)	%token TOKEN
	TOKEN is used in production

The external variable yylval

- is used in a LEX source program to return values of lexemes,
- yylval is assumed to be integer if you take no other action.
- Changes related to yylval must be made
  - in the definitions section of YACC specification
    - by adding new types in the following way
    - %union {
   (type fieldname)
   (type fieldname)
   ....
}
    - and defining which token and non-terminals will use these types
    - %token <fieldname> token
%type <fieldname> non-terminal

If you need a record type, then add it in the union. Example:

```
%union {
  struct s {
    double fvalue;
    int ivalue;
  } t;
}
```

Design of Program :

Code : ParserCode.y

```
%union{

      char *intval;
      char *floatval;
      char *stringval;
      struct SymbTab *symp;
}
%{
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>
#include "main.c"
extern FILE* yyin;
extern FILE* yyout;

extern char *temp;
extern  int i,j;
extern  int count ;
int yylineno;
int yyerror(char *);
int yylex(void);
%}

/* Decleration*/
%token PREPROCESSOR HEADER KEYWORDS LINE SPACE COMMA LESS VOID S_ADD INT CHAR FLOAT FOR QUOT
%token OPENBC CLOSEBC POINTER ARRAY DEFINE CCBRACE OCBRACE MAIN S_SUB VARCHAR PRINTF
%token ASSIGNMENT PLUS MINUS MULTIPLY DIVIDE MODULO INCREMENT DECREMENT S_MUL
```


```
/* Decleration*/
%token PREPROCESSOR HEADER KEYWORDS LINE SPACE COMMA LESS VOID S_ADD INT CHAR FLOAT FOR QUOT
%token OPENBC CLOSEBC POINTER ARRAY DEFINE CCBRACE OCBRACE MAIN S_SUB VARCHAR PRINTF
%token ASSIGNMENT PLUS MINUS MULTIPLY DIVIDE MODULO INCREMENT DECREMENT S_MUL
```


```
%token ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN WHILE RETURN S_DIV
%token SEMICOLON IF ELSE LESS_EQUAL MORE_EQUAL MORE EQUAL NOT_EQUAL
%left PLUS MINUS
%left MULTIPLY DIVIDE



%token <intval> DIGIT
%token <floatval> FDIGIT
%token <stringval> STRING


%%
    /* Rules*/

    program
            : program funcdef
            | funcdef
            | pre_pro program
            ;
    pre_pro
            : PREPROCESSOR HEADER
            ;
    funcdef
```

```
pre_pro
        : PREPROCESSOR HEADER
        ;
funcdef
        : VOID MAIN args block_statement
        | types VARCHAR args block_statement
        ;


args
        :OCBRACE var_def_list CCBRACE
        ;

var_def_list
        :
        | var_def
        | var_def var_def_list_extension
        ;

var_def_list_extension
        : COMMA var_def
        ;

var_def
        : types VARCHAR
        ;

types
```

```
types
        : INT
        | CHAR
        | FLOAT
        ;

block_statement
        : OPENBC statements CLOSEBC
        ;

statements
        :
        | statement statements
        ;

statement
        : block_statement
        | print_statement
        | conditional_statement
        | while_st
        | for_st
        | assignment_statement SEMICOLON
        | declaration_statement SEMICOLON
        | function_call SEMICOLON
        | ret_statement SEMICOLON
        ;

print_statement
        : PRINTF OCBRACE STRING CCBRACE SEMICOLON
```

```
 print_statement
         : PRINTF OCBRACE STRING CCBRACE SEMICOLON
         ;

 declaration_statement
         : types id_list
         ;

id_list
         : VARCHAR id_list_1
         ;
id_list_1
         : COMMA VARCHAR id_list_1
         |
         ;

 function_call
         : VARCHAR OCBRACE expression_list CCBRACE
         ;

 conditional_statement
         : IF OCBRACE conditions CCBRACE block_statement elsest
         ;

 elsest
         : ELSE block_statement
         |
         ;
```

```
  elsest
         : ELSE block_statement
         |
         ;

 while_st
         : WHILE OCBRACE conditions CCBRACE block_statement
         ;

 for_st
         : FOR OCBRACE INT VARCHAR ASSIGNMENT DIGIT SEMICOLON conditions SEMICOLON _inc_decre CCBRACE block
         |for_woin
         |for_wocon
         |for_wodec
         ;

for_wodec
         : FOR OCBRACE SEMICOLON conditions SEMICOLON _inc_decre CCBRACE block_statement
         ;

for_wocon
         : FOR OCBRACE SEMICOLON SEMICOLON _inc_decre CCBRACE block_statement
         ;

for_woin
         : FOR OCBRACE INT VARCHAR SEMICOLON conditions SEMICOLON  CCBRACE block_statement
         ;

 _inc_decre
```

```
for_woin
          : FOR OCBRACE INT VARCHAR SEMICOLON conditions SEMICOLON  CCBRACE block_statement
          ;

_inc_decre
          :_inc
          |_dec
          ;
_inc
          : VARCHAR INCREMENT
          | INCREMENT VARCHAR
          ;
_dec
          : VARCHAR DECREMENT
          | DECREMENT VARCHAR
          ;

 conditions
          : int_cond
          ;


int_cond
          : int_cond LESS int_expression
          | int_cond LESS_EQUAL int_expression
          | int_cond MORE_EQUAL int_expression
          | int_cond MORE int_expression
          | int_cond NOT_EQUAL int_expression
```

```
int_cond
          : int_cond LESS int_expression
          | int_cond LESS_EQUAL int_expression
          | int_cond MORE_EQUAL int_expression
          | int_cond MORE int_expression
          | int_cond NOT_EQUAL int_expression
          | int_cond EQUAL int_expression
          | int_expression
          ;


assignment_statement
            : int_ass
            | char_ass
            | types VARCHAR ASSIGNMENT {  }
            | error
            ;
  int_ass
            :types VARCHAR ASSIGNMENT int_expression
            | VARCHAR ASSIGNMENT int_expression
            ;
  char_ass
            :types VARCHAR ASSIGNMENT char_expression
            | VARCHAR ASSIGNMENT char_expression
            ;

ret_statement
            : RETURN int_expression {  }
```

```
  ret_statement
            : RETURN int_expression {  }
            ;
  int_expression
              : DIGIT
              | VARCHAR
              | int_expression PLUS int_expression
              | int_expression MINUS int_expression
              | int_expression MULTIPLY int_expression
              | int_expression DIVIDE int_expression
              | OCBRACE int_expression CCBRACE
              ;

  char_expression
              : QUOT STRING QUOT { }
              ;

  expression_list
              : expression_list COMMA int_expression
              | expression_list COMMA STRING
              | int_expression
              | STRING
              ;



%%

int yyerror(char *s) {
```

```
  expression_list
              : expression_list COMMA int_expression
              | expression_list COMMA STRING
              | int_expression
              | STRING
              ;



%%

int yyerror(char *s) {
    fprintf(stderr , "%s\n", s);
    exit(0);
}

int main()
{
  yyin = fopen("program2.txt","r");
  FILE *file= fopen("Symbol_Table.txt", "w");
  yyout= file;

  yyparse();
  printf("\nProgram parsed successfully.\n");
  Display();
  fclose(yyout);
  fclose(yyin);
}
```

Test Cases :

Without Error : test_Case.c

```
// test case 5
// NO ERRORS

#include<stdio.h>
void main ()
{
	printf("\n this program has no errors");
}
```

Output :

1. Console Output -

```
line number : 1 : // test case 5 : is a comment
line number : 2 : // NO ERRORS : is a comment
line number : 4 : #include is a preprocessor
line number : 4 : <stdio.h> is a header
line number : 5 : C Keyword(1) : void
line number : 5 : C Keyword(2) : main
line number : 5 : open curly brace (
line number : 5 : closing curly brace )
line number : 6 : open round brace {
line number : 7 : C Keyword(3) : printf
line number : 7 : open curly brace (
line number : 7 : \n this program has no errors , is a string
line number : 7 : closing curly brace )
line number : 7 : semi colon ;
line number : 8 : close round brace }
Program parsed successfully.
```

2. Symbol Table -

SYMBOL	TOKEN	Attribute Number
#include	preprocessor	0
<stdio.h>	header	1
void	keywords	2
main	keywords	3
(	Open_Brace	4
)	Close_Brase	5
{	Open_Paran	6
printf	keywords	7
(	Open_Brace	4
\n this program has no errors	String	8
)	Close_Brase	5
;	SemiColon	9
}	Close_Paran	10

Test Cases With Error :

test_Case1.c (Missing Closing Parenthesis)

```c
// Test case 1
// missing parenthesis


#include<stdio.h>
void main ()
{
        int x = 10,y;
        while(x>0)
        {
                x--;
                        // ERROR : Missing closing Parenthesis
}
```

Output -

```
line number : 1 : // Test case 1 : is a comment

line number : 2 : // missing parenthesis : is a comment

line number : 5 : #include is a preprocessor

line number : 5 : <stdio.h> is a header

line number : 6 : C Keyword(1) : void

line number : 6 : C Keyword(2) : main

line number : 6 : open curly brace (

line number : 6 : closing curly brace )

line number : 7 : open round brace {

line number : 8 : C Keyword(3) : int

line number : 8 : x is a variable name

line number : 8 : equal to operator =

line number : 8 : 10 is a integer

line number : 8 : comma ,
syntax error
```

test_Case2.c (Missing Semicolon)

```c
// test case 2
// missing semicolon


#include<stdio.h>
void main ()
{
        int x,y;
        x = x + y       // missing semicolon
        printf("%d",x);
}
```

Output -

```
line number : 1 : // test case 2 : is a comment

line number : 2 : // missing semicolon : is a comment

line number : 5 : #include is a preprocessor

line number : 5 : <stdio.h> is a header

line number : 6 : C Keyword(1) : void

line number : 6 : C Keyword(2) : main

line number : 6 : open curly brace (

line number : 6 : closing curly brace )

line number : 7 : open round brace {

line number : 8 : C Keyword(3) : int

line number : 8 : x is a variable name

line number : 8 : comma ,

line number : 8 : y is a variable name

line number : 8 : semi colon ;

line number : 9 : x is a variable name

line number : 9 : equal to operator =

line number : 9 : x is a variable name

line number : 9 : addition operator +

line number : 9 : y is a variable name

line number : 9 : // missing semicolon : is a comment

line number : 10 : C Keyword(4) : printf
syntax error
```

test_Case3.c (Undeclared Variable)**-**

The identification of undeclared variable is the functionality of Semantic Analysis phase of compiler design. Therefore, this test case will be parsed successfully because parser won't be able to recognize this error.

```
// test case 3
// undeclared variable

#include<stdio.h>
void main ()
{
        int x,y;
        z = x + y; // undeclared value z
}
```

Output -

```
line number : 1 : // test case 3 : is a comment
line number : 2 : // undeclared variable : is a comment
line number : 4 : #include is a preprocessor
line number : 4 : <stdio.h> is a header
line number : 5 : C Keyword(1) : void
line number : 5 : C Keyword(2) : main
line number : 5 : open curly brace (
line number : 5 : closing curly brace )
line number : 6 : open round brace {
line number : 7 : C Keyword(3) : int
line number : 7 : x is a variable name
line number : 7 : comma ,
line number : 7 : y is a variable name
line number : 7 : semi colon ;
line number : 8 : z is a variable name
line number : 8 : equal to operator =
line number : 8 : x is a variable name
line number : 8 : addition operator +
line number : 8 : y is a variable name
line number : 8 : semi colon ;
line number : 8 : // undeclared value z : is a comment
```

```
line number : 8 : // undeclared value z : is a comment
line number : 9 : close round brace }
Program parsed successfully.
```

test_Case4.c (Unbalanced Expression)

```
// test case 4
// unbalenced expressions

#include<stdio.h>
void main ()
{
	int x,y;
	float z;
	z = (((x+y)/(x-y)); // unbalenced expression
}
```

Output-

```
line number : 1 : // test case 4 : is a comment
line number : 2 : // unbalenced expressions : is a comment
line number : 4 : #include is a preprocessor
line number : 4 : <stdio.h> is a header
line number : 5 : C Keyword(1) : void
line number : 5 : C Keyword(2) : main
line number : 5 : open curly brace (
line number : 5 : closing curly brace )
line number : 6 : open round brace {
line number : 7 : C Keyword(3) : int
line number : 7 : x is a variable name
line number : 7 : comma ,
line number : 7 : y is a variable name
line number : 7 : semi colon ;
line number : 8 : C Keyword(4) : float
line number : 8 : z is a variable name
line number : 8 : semi colon ;
line number : 9 : z is a variable name
line number : 9 : equal to operator =
line number : 9 : open curly brace (
line number : 9 : open curly brace (
```

```
line number : 9 : open curly brace (
line number : 9 : x is a variable name
line number : 9 : addition operator +
line number : 9 : y is a variable name
line number : 9 : closing curly brace )
line number : 9 : division operator /
line number : 9 : open curly brace (
line number : 9 : x is a variable name
line number : 9 : subtraction operator -
line number : 9 : y is a variable name
line number : 9 : closing curly brace )
line number : 9 : closing curly brace )
line number : 9 : semi colon ;
syntax error
```

One test case for all (No Errors)

This is one special test case which checks all the functionalities.

```
4 #include<stdio.h>
5 void main ()
6 {
7         int x = -10 , y = '1' , z , a ;
8
9         for ( int i =0 ; i<10 ; ++i ) {
10                                for ( int j = 0,k = 19; j < k ; j++,k--)
11                                        {
12                                }
13                }
14        for ( ; ; )
15        for ( ; ; );
16        while(1);
17
18 }
19
20 int abc () { }
```

Output -

```
line number : 4 : #include is a preprocessor
line number : 4 : <stdio.h> is a header
line number : 5 : C Keyword(1) : void
line number : 5 : C Keyword(2) : main
line number : 5 : open curly brace (
line number : 5 : closing curly brace )
line number : 6 : open round brace {
line number : 7 : C Keyword(3) : int
line number : 7 : x is a variable name
line number : 7 : equal to operator =
line number : 7 : comma ,
line number : 7 : y is a variable name
line number : 7 : equal to operator =
line number : 7 : 1 , is a string
line number : 7 : comma ,
line number : 7 : z is a variable name
line number : 7 : comma ,
line number : 7 : a is a variable name
line number : 7 : semi colon ;
line number : 9 : C Keyword(4) : for
```

```
line number : 9 : open curly brace (
line number : 9 : C Keyword(5) : int
line number : 9 : i is a variable name
line number : 9 : equal to operator =
line number : 9 : 0 is a integer
line number : 9 : semi colon ;
line number : 9 : i is a variable name
line number : 9 : less than operator <
line number : 9 : 10 is a integer
line number : 9 : semi colon ;
line number : 9 : increment operator ++
line number : 9 : i is a variable name
line number : 9 : closing curly brace )
line number : 9 : open round brace {
line number : 10 : C Keyword(6) : for
line number : 10 : open curly brace (
line number : 10 : C Keyword(7) : int
line number : 10 : j is a variable name
line number : 10 : equal to operator =
line number : 10 : 0 is a integer
line number : 10 : comma ,
```

```
line number : 10 : k is a variable name
line number : 10 : equal to operator =
line number : 10 : 19 is a integer
line number : 10 : semi colon ;
line number : 10 : j is a variable name
line number : 10 : less than operator <
line number : 10 : k is a variable name
line number : 10 : semi colon ;
line number : 10 : j is a variable name
line number : 10 : increment operator ++
line number : 10 : comma ,
line number : 10 : k is a variable name
line number : 10 : decrement operator --
line number : 10 : closing curly brace )
line number : 11 : open round brace {
line number : 12 : close round brace }
line number : 13 : close round brace }
line number : 14 : C Keyword(8) : for
line number : 14 : open curly brace (
line number : 14 : semi colon ;
line number : 14 : semi colon ;
```

```
line number : 14 : closing curly brace )
line number : 15 : C Keyword(9) : for
line number : 15 : open curly brace (
line number : 15 : semi colon ;
line number : 15 : semi colon ;
line number : 15 : closing curly brace )
line number : 15 : semi colon ;
line number : 16 : C Keyword(10) : while
line number : 16 : open curly brace (
line number : 16 : 1 is a integer
line number : 16 : closing curly brace )
line number : 16 : semi colon ;
line number : 18 : close round brace }
line number : 20 : C Keyword(11) : int
line number : 20 : abc is a variable name
line number : 20 : open curly brace (
line number : 20 : closing curly brace )
line number : 20 : open round brace {
line number : 20 : close round brace }
Program parsed successfully.
```

Implementation :

The Productions for most of the features of C are fairly straightforward. A few important ones are :

Compound Statement:

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

Selection Statement:

```
selection_statement
                    : IF '(' expression ')' statement
                    | IF '(' expression ')' statement ELSE statement
                    | SWITCH '(' expression ')' statement
                    ;
```

Iteration Statement:

```
iteration_statement
          : WHILE '(' expression ')' statement
          | FOR '(' expression_statement expression_statement ')' statement
          | FOR '(' expression_statement expression_statement expression ')' statement
          ;
```

Results :

The lex (LexerCode.l) and yacc (ParserCode.y) codes are compiled and executed by the following terminal commands to parse the input test file.

```
lex LexerCode.l
yacc -d ParserCode.y
cc y.tab.c lex.yy.c -ll -ly
./a.out
```

After parsing, if there are errors then a ‘syntax error’ message will be displayed on the terminal. Otherwise, a ‘Program parsed successfully’ message is displayed on the console.

Handling Conflicts :

Shift-Reduce Conflict -

The Shift-Reduce Conflict is the most common type of conflict found in grammars. It is caused when the grammar allows a rule to be reduced for particular token, but, at the same time, allowing another rule to be shifted for that same token. As a result, the grammar is ambiguous since a program can be interpreted more than one way. This error is often caused by recursive grammar definitions where the system cannot determine when one rule is complete and another is just started.

Reduce-Reduce Conflict -

A Reduce-Reduce error is a caused when a grammar allows two or more different rules to be reduced at the same time, for the same token. When this happens, the grammar becomes ambiguous since a program can be interpreted more than one way. This error can be caused when the same rule is reached by more than one path.

Removing Conflicts -

Once we have figured out the actual reason for a conflict, we can try and modify the grammar to remove the conflict without changing the overall language. We can do this either by changing the grammar to remove the source of the conflict, or give yacc additional information to allow it to figure out which of the conflicting actions is the right one to choose, or (if we know what we're doing) we can leave the conflict alone and rely
on yacc's default action. It's important to realize that conflicts cannot always be removed: there are some grammars for which, no matter what we do, we cannot remove all conflicts while preserving equivalence of grammars. The approaches outlined below are therefore heuristics that often work, though they may not always do so.

Solving dangling else problem:

There is a much simpler solution. If you know how LR parsers work, then you know that the conflict happens here:

IF (expression) statement * ELSE statement where the star marks the current position of the cursor. The question the parser must answer is "should I shift, or should I reduce". Usually, you want to bind the else to the closest if, which means you want to shift the else token now. Reducing now would mean that you want the else to wait to be bound to an older if.

We should specify the parser generator that "when there is a shift/reduce conflict between the token ELSE and the rule "selection_statement -> IF (expression) statement", then the token must win". To do so, a name is given to the precedence of your rule (e.g., NO_ELSE), and specify that NO_ELSE has less precedence than ELSE. Something like:

```
//Precedences go increasing, So, NO_ELSE < ELSE
%nonassoc NO_ELSE
%nonassoc ELSE
%%
selection_statement
        : IF '(' expression ')' statement                    %prec NO_ELSE
        | IF '(' expression ')' statement ELSE statement
        ;
```

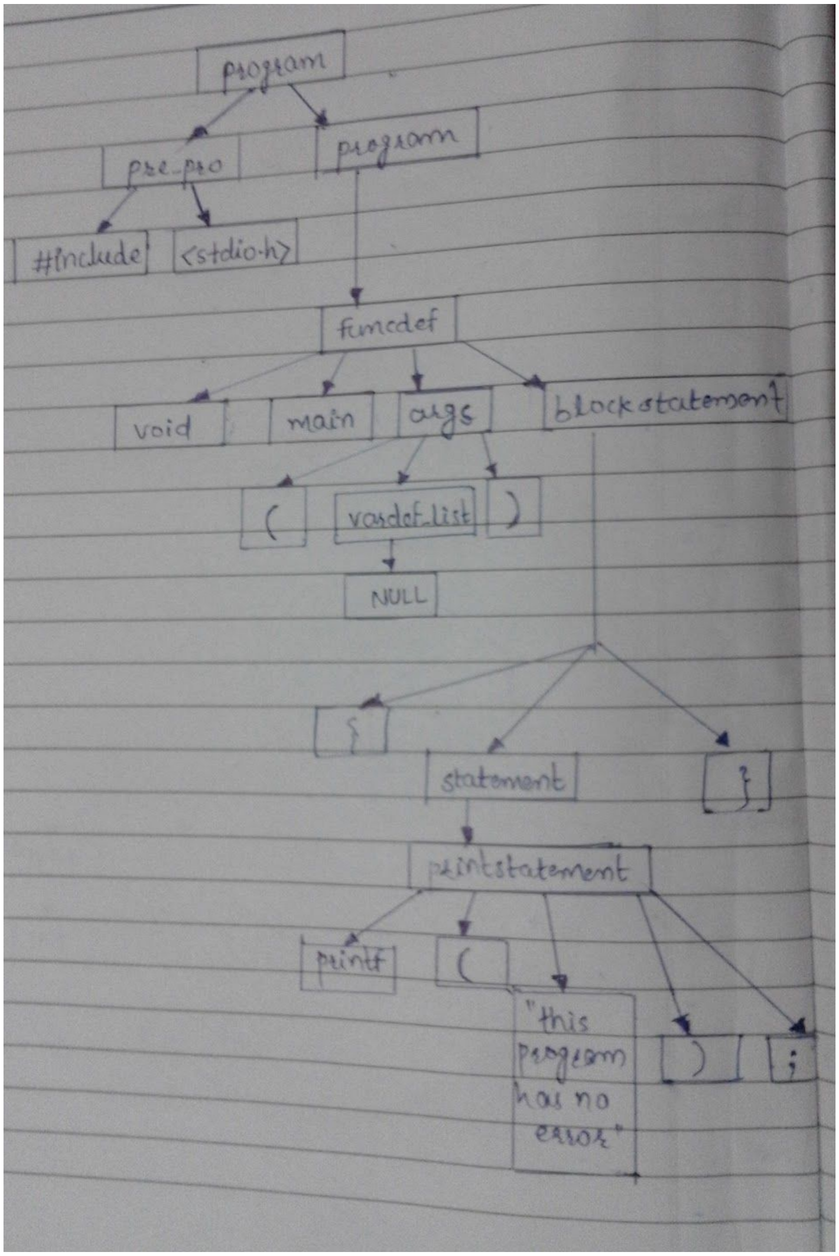
Precedences:

Yacc allows you to specify these choices with the operator precedence declarations %left and %right. Each such declaration contains a list of tokens, which are operators whose precedence and associativity is being declared. The %left declaration makes all those operators left-associative and the %right declaration makes them right-associative. A third alternative is %nonassoc, which declares that it is a syntax error to find the same operator twice “in a row”. The last alternative,%precedence, allows to define only precedence and no associativity at all. The directive %nonassoc creates run-time error: using the operator in a associative way is a syntax error. The directive %precedence creates compile-time errors.

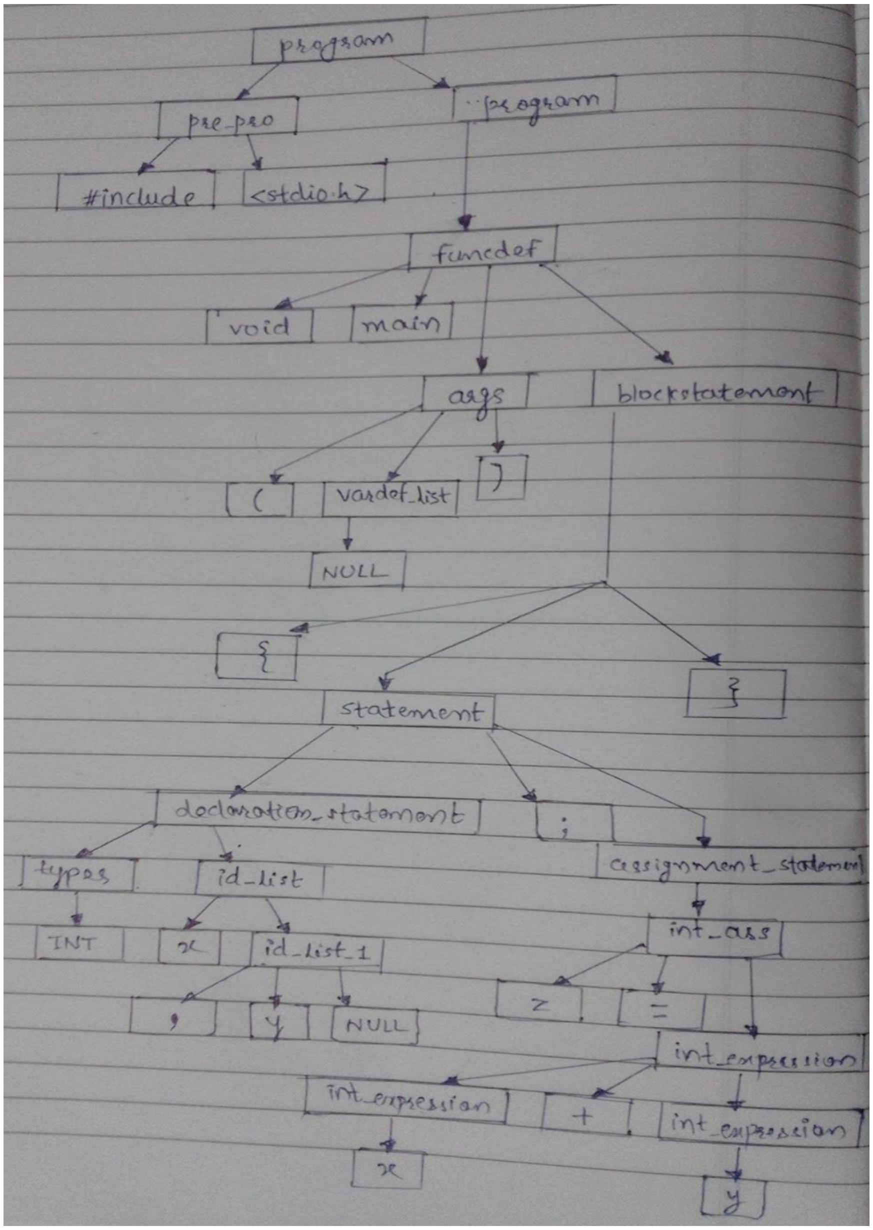
The relative precedence of different operators is controlled by the order in which they are declared. The first precedence/associativity declaration in the file declares the operators whose precedence is lowest, the next such declaration declares the operators whose precedence is a little higher, and so on.
We have followed the C precedence and associativity order in our productions.

Parse Trees -

1. test_Case.c (No Errors)



2. test_Case3.c (Undeclared Variable) -



Future Work :

In the future weeks we plan to implement more features considering it might be required for the semantic analysis phase implementation. We have removed all the Shift-Reduce and Reduce-Reduce conflicts from our code.

Conclusion :

This report briefs about the Syntax Analysis Phase of project. The parser code is provided in the report along with the necessary test cases and their respective outputs. Also we give the exposure to the different types of parsing. We provide the insight of Yacc script also. We will continue to modify the code in future based on the requirement of implementation of further phases of compiler design.