National Institute of Technology Karnataka Surathkal

Date: 20 February 2018
Submitted To: Ms. Sushmita

Project 2: PARSER FOR C LANGUAGE

Group Members:

Roll No	Name	E-mail	Contact No.
15CO230	Naladala Indukala	15co230.indukala@nitk.edu.in	9880692703
15CO236	R Aparna	15co236.aparna@nitk.edu.in	8105348941

ABSTRACT

A compiler is a computer program that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages or passes, builds the output code. Syntax analysis or parsing is the second phase of a compiler. 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 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. 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. It parses the entire code even if there are errors in the code. This is done using error recovery techniques. Parsing, syntax analysis or syntactic analysis is the process of analysing a string of symbols, either in natural language or in computer languages, conforming to the rules of a formal grammar. Syntactic analysis, or parsing, is needed to determine if the series of tokens given are appropriate in a language - that is, whether or not the sentence has the right shape/form. However, not all syntactically valid sentences are meaningful, further semantic analysis has to be applied for this. For syntactic analysis, context-free grammars and the associated parsing techniques are powerful enough to be used - this overall process is called parsing. There are many techniques for parsing algorithms, and the two main classes of algorithm are top-down and bottom-up parsing.

TABLE OF CONTENTS

1. INTRODUCTION

Syntax Analysis

In computer science, syntax analysis is the process of checking that the code is syntactically correct. The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are valid sequence of symbols, keywords, identifiers etc. The parser needs to be able to handle the infinite number of possible valid programs that may be presented to it. The usual way to define the language is to specify a *grammar*. A grammar is a set of rules (or *productions*) that specifies the syntax of the language (i.e. what is a valid sentence in the language). There can be more than one grammar for a given language.

Flex Script

FLEX (Fast LEXical analyzer generator) is a tool for generating scanners. Instead of writing a scanner from scratch, you only need to identify the *vocabulary* of C language and write a specification of patterns using regular expressions and FLEX will construct a scanner for you.

The structure of our flex script is intentionally similar to that of a Yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

Definition section
%%
Rules section
%%
C code section

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

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

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

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

It may contain the following items:

- Declarations of tokens: Yacc requires token names to be declared 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:

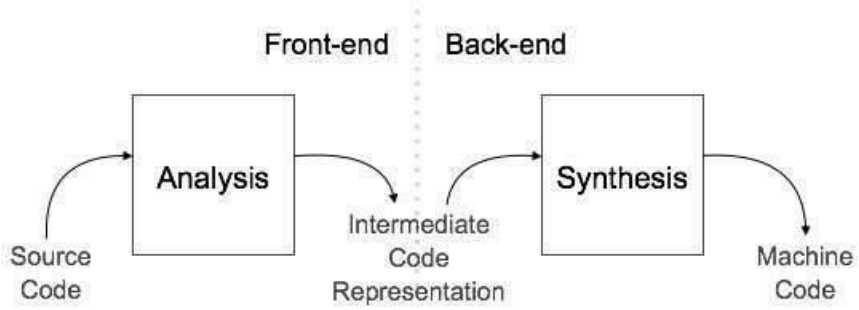
nonterminal: sentential form
| sentential form
.....
| sentential form
;

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

2. DESIGN OF PROGRAM

This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned into account. Finally, a file called lex.yy.c is generated, which when executed recognizes the tokens present in the C program which was given as input.
The yacc script is run along with the lex script to parse the C code given as input and specify the errors (if any) or tell the user that the parsing has been successfully completed.

Flow :



Parser checks the C code for syntactical errors and specifies them if present. Otherwise, it displays 'Parsing Complete'.

Codes:

Lex Code: (scanner.l file)

```
alpha [A-Za-z_]
```

```
digit [0-9]
space [ ]

%{
int yylineno;
#include <stdio.h>
struct symbol
{
char values[100];
char type[15];
};

int k=0, j=0, i=0;
typedef struct symbol s;
s ident[100];
s constant[100];

void insert(int a, char val[], char type[])
{
if(a==0)
{
for(k=0; k<j; k++)
{
if(strcmp(val, constant[k].values)==0)
return;
}
strcpy(constant[j].values, val);
strcpy(constant[j].type, type);
j++;
}
else if(a==1)
{
for(k=0; k<i; k++)
{
if(strcmp(val, ident[k].values)==0)
return;
}
strcpy(ident[i].values, val);
strcpy(ident[i].type, type);
```

```
i++;
}
}
%}

%%
\n { yylineno++; }
"/*" { multicomment(); }
"//" { singlecomment(); }
"#include<"({alpha})*".h>" {}
"#define"({space})""({alpha})""({alpha}|{digit})*""({space})""({digit})+""
{ return DEF;}
{digit}+ { insert(0,yytext,"Integer"); return CONSTANT; }
({digit}+)\.({digit}+) { insert(0,yytext,"Float"); return CONSTANT; }
{alpha}?\"(\\.|[^\\"])*\"  {  insert(0,yytext,"String  Literal");  return
STRING_LITERAL; }
"sizeof" { return SIZEOF; }
"->" { return PTR_OP; }
"++" { 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_SHORT; }
"/=" { return DIV_SHORT; }
"%=" { return MOD_SHORT; }
"+=" { return ADD_SHORT; }
"-=" { return SUB_SHORT; }
"<<=" { return LEFT_SHORT; }
">>=" { return RIGHT_SHORT; }
"&=" { return AND_SHORT; }
"^=" { return XOR_SHORT; }
"|=" { return OR_SHORT; }
```

```
"typedef" { return TYPEDEF; }
"extern" { return EXTERN; }
"static" { return STATIC; }
"auto" { return AUTO; }
"register" { return REGISTER; }
"char" { return CHAR; }
"int" { return INT; }
"float" { return FLOAT; }
"const" { return CONST; }
"volatile" { return VOLATILE; }
"void" { return VOID; }
"struct" { return STRUCT; }
"union" { return UNION; }
"enum" { return ENUM; }
"case" { return CASE; }
"default" { return DEFAULT; }
"if" { return IF; }
"else" { return ELSE; }
"switch" { return SWITCH; }
"while" { return WHILE; }
"do" { return DO; }
"for" { return FOR; }
"goto" { return GOTO; }
"continue" { return CONTINUE; }
"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('?'); }
"printf" {insert(1,yytext,"Function");return IDENTIFIER; }
"main" {insert(1,yytext,"Function");return IDENTIFIER; }
{alpha}({alpha}|{digit})* {insert(1,yytext,"Identifier");return IDENTIFIER;
}
[ \t\v\n\f] { }
. { /* ignore any other illegal characters */ }
%%
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);
}
```

Yacc Code: (parser.y file)

```
%expect 24

%nonassoc NO_ELSE
%nonassoc ELSE
%left '<' '>' '=' GE_OP LE_OP EQ_OP NE_OP
%left '+' '-'
%left '*' '/' '%'
%left '|'
%left '&'
%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_SHORT DIV_SHORT MOD_SHORT ADD_SHORT
%token SUB_SHORT LEFT_SHORT RIGHT_SHORT AND_SHORT
%token XOR_SHORT OR_SHORT TYPE_NAME DEF
%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM
%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN
%start translation_unit
%%

Define
: DEF ;

primary_exp
: IDENTIFIER
| CONSTANT
```

```
| STRING_LITERAL
| '(' expression ')'
| Define primary_exp
;

postfix_exp
: primary_exp
| postfix_exp INC_OP
| postfix_exp DEC_OP
| postfix_exp PTR_OP IDENTIFIER
| postfix_exp '(' argument_expression_list ')'
| postfix_exp '.' IDENTIFIER
| postfix_exp '[' expression ']'
| postfix_exp '(' ')'
;

argument_expression_list
: assignment_expression
| argument_expression_list ',' assignment_expression
;

unary_expression
: postfix_exp
| INC_OP unary_expression
| DEC_OP unary_expression
| unary_op cast_expression
| SIZEOF unary_expression
| SIZEOF '(' type_name ')'
;

unary_op
: '&'
| '*'
| '+'
| '-'
| '~'
| '!'
;
```

```
cast_expression
: unary_expression
| '(' type_name ')' cast_expression
;

multiplicative_expression
: cast_expression
| multiplicative_expression '*' cast_expression
| multiplicative_expression '/' cast_expression
| multiplicative_expression '%' cast_expression
;

additive_expression
: multiplicative_expression
| additive_expression '+' multiplicative_expression
| additive_expression '-' multiplicative_expression
;

shift_expression
: additive_expression
| shift_expression LEFT_OP additive_expression
| shift_expression RIGHT_OP additive_expression
;

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

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

and_expression
: equality_expression
```

```
| and_expression '&' equality_expression
;

exclusive_or_expression
: and_expression
| exclusive_or_expression '^' and_expression
;

inclusive_or_expression
: exclusive_or_expression
| inclusive_or_expression '|' exclusive_or_expression
;

logical_and_expression
: inclusive_or_expression
| logical_and_expression AND_OP inclusive_or_expression
;

logical_or_expression
: logical_and_expression
| logical_or_expression OR_OP logical_and_expression
;

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

assignment_expression
: conditional_expression
| unary_expression assign_op assignment_expression
;

assign_op
: '='
| MUL_SHORT
| DIV_SHORT
| MOD_SHORT
| ADD_SHORT
| SUB_SHORT
| LEFT_SHORT
```

```
| RIGHT_SHORT
| AND_SHORT
| XOR_SHORT
| OR_SHORT
;
expression
: assignment_expression
| expression ',' assignment_expression
;
constant_expression
: conditional_expression
;
declaration
: declaration_specifiers ';'
| declaration_specifiers init_declarator_list ';'
| error
;
declaration_specifiers
: storage_class
| storage_class declaration_specifiers
| data_type
| data_type declaration_specifiers
| type_qualifier
| type_qualifier declaration_specifiers
;
init_declarator_list
: init_declarator
| init_declarator_list ',' init_declarator
;
init_declarator
: declarator
| declarator '=' initializer
;
storage_class
```

```
: REGISTER
| EXTERN
| TYPEDEF
| AUTO
| STATIC
;

data_type
: VOID
| CHAR
| SHORT
| INT
| LONG
| FLOAT
| DOUBLE
| SIGNED
| UNSIGNED
| struct_or_union_specifier
| enum_specifier
| TYPE_NAME
;

struct_or_union_specifier
: struct_or_union IDENTIFIER '{' struct_declaration_list '}'
| struct_or_union '{' struct_declaration_list '}'
| struct_or_union IDENTIFIER
;

struct_or_union
: STRUCT
| UNION
;

struct_declaration_list
: struct_declaration
| struct_declaration_list struct_declaration
;

struct_declaration
: specifier_qualifier_list struct_declarator_list ';'
```

```
;

specifier_qualifier_list
: data_type specifier_qualifier_list
| data_type
| type_qualifier specifier_qualifier_list
| type_qualifier
;

struct_declarator_list
: struct_declarator
| struct_declarator_list ',' struct_declarator
;

struct_declarator
: declarator
| ':' constant_expression
| declarator ':' constant_expression
;

enum_specifier
: ENUM '{' enumerator_list '}'
| ENUM IDENTIFIER '{' enumerator_list '}'
| ENUM IDENTIFIER
;

enumerator_list
: enumerator
| enumerator_list ',' enumerator
;

enumerator
: IDENTIFIER
| IDENTIFIER '=' constant_expression
;

type_qualifier
: CONST
| VOLATILE
;

declarator
```

```
: pointer direct_declarator
| direct_declarator
;

direct_declarator
: IDENTIFIER
| '(' declarator ')'
| direct_declarator '[' constant_expression ']'
| direct_declarator '[' ']'
| direct_declarator '(' parameter_type_list ')'
| direct_declarator '(' identifier_list ')'
| direct_declarator '(' ')'
;

pointer
: '*'
| '*' type_qualifier_list
| '*' pointer
| '*' type_qualifier_list pointer
;

type_qualifier_list
: type_qualifier
| type_qualifier_list type_qualifier
;

parameter_type_list
: parameter_list
;

parameter_list
: parameter_declaration
| parameter_list ',' parameter_declaration
;

parameter_declaration
: declaration_specifiers declarator
| declaration_specifiers abstract_declarator
| declaration_specifiers
;
```

```
identifier_list
: IDENTIFIER
| identifier_list ',' IDENTIFIER
;

type_name
: specifier_qualifier_list
| specifier_qualifier_list abstract_declarator
;

abstract_declarator
: pointer
| direct_abstract_declarator
| pointer direct_abstract_declarator
;

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

initializer
: assignment_expression
| '{' initializer_list '}'
| '{' initializer_list ',' '}'
;

initializer_list
: initializer
| initializer_list ',' initializer
;

statement
```

```
: labeled_statement
| compound_statement
| expression_statement
| selection_statement
| loop_stmt
| jump_statement
;

labeled_statement
: IDENTIFIER ':' statement
| CASE constant_expression ':' statement
| DEFAULT ':' statement
;

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

declaration_list
: declaration
| declaration_list declaration
;

statement_list
: statement
| statement_list statement
;

expression_statement
: ';'
| expression ';'
;

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

```
;

loop_stmt
: WHILE '(' expression ')' statement
| DO statement WHILE '(' expression ')' ';'
| FOR '(' expression_statement expression_statement ')' statement
| FOR '(' expression_statement expression_statement expression ')'
statement
;

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

translation_unit
: external_declaration
| translation_unit external_declaration
| Define translation_unit
;

external_declaration
: function_definition
| declaration
;

function_definition
: declaration_specifiers declarator declaration_list compound_statement
| declaration_specifiers declarator compound_statement
| declarator declaration_list compound_statement
| declarator compound_statement
;

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

```
void print_constant()
{
printf("\n___________CONSTANTS TABLE___________\n");
printf("\nName\t\t\tType\n");
for(k=0; k<j; k++)
printf("%s\t\t\t%s\n", constant[k].values, constant[k].type);
printf("\n");
}

void print_symbols()
{
printf("\n___________SYMBOLS TABLE_____________\n");
printf("\nName\t\t\tType\n");
for(k=0; k<i; k++)
printf("%s\t\t\t%s\n", ident[k].values, ident[k].type);
printf("\n\n\n");
}

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

if(!yyparse())
{
print_constant();
print_symbols();
printf("\nParsing complete\n");
}
else
printf("\nParsing failed\n");

fclose(yyin);
return 0;
}
extern char *yytext;

yyerror(char *s)
{
printf("\nLine %d : %s\n", (yylineno), s);
}
```

3. EXPLANATION OF PROGRAM

Files :

- **Scanner.l**: Lex file which defines all the terminals of the productions stated in the Yacc file. It contains regular expressions as well as functions to handle single and multi-line comments in the C code given as input.
- **Parser.y**: Yacc file is where the productions for all the loops, selection and conditional statements and expressions are mentioned.
- **Testcase_1.txt:** The input C code which will be parsed by executing the lex and yacc files along with it.

The flex script recognises the following classes of tokens from the input:

- Pre-processor instructions
  Statements processed: *#include<stdio.h>, #define var1 var2*
  Token generated: Header / Preprocessor Directive

- Literals
  Statements processed: *int, float*
  Tokens generated: Keyword

- Keywords
  Statements processed: *if, else, void, while, do, int, float, break, return* and so on.
  Tokens generated: Keyword

- Identifiers
  Statements processed: *a, abc, a_b, a12b4*
  Tokens generated: Identifier

- Constants
  Statements processed: 123, 1.23, “Hi”
  Tokens generated: Constants

The Yacc script specifies productions for the following:

- Primary expressions
- Argument expressions list
- Unary expressions
- Postfix expressions
- Operators
- Unary operators
- Cast expressions
- Multiplicative expressions
- Additive expressions
- Shift expressions
- Relational expressions

- Equality expressions
- AND expressions
- Exclusive OR expressions
- Inclusive OR expressions
- Logical AND expressions
- Logical OR expressions
- Conditional expressions
- Assignment expressions
- Declarations
- Storage Class Specifiers
- Type Specifier
- Structures
- Unions
- Enum
- Pointer declarations
- Abstract declarations
- Parameter and Identifier Lists
- Labelled statement
- Compound statement
- Selection statement
- Iteration statement
- Jump statement
- Function definitions
- Parentheses (all types)
- Arrays

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division(modulo division)

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 returns 0
>	Greater than	5 > 3 returns 1
<	Less than	5 < 3 returns 0
!=	Not equal to	5 != 3 returns 1
>=	Greater than or equal to	5 >= 3 returns 1
<=	Less than or equal to	5 <= 3 return 0

Operator	Meaning of Operator	Example
&&	Logial AND. True only if all operands are true	If c = 5 and d = 2 then, expression `((c == 5) && (d > 5))` equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression `((c == 5) || (d > 5))` equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression `! (c == 5)` equals to 0.

Information about Code

- %expect 24
  It indicates that there should be 24 shift/reduce conflicts and no reduce/reduce conflicts. Bison reports an error if the number of shift/reduce conflicts differs from 24 or if there are any reduce/reduce conflicts.

- The start symbol for the grammar is “translation_unit”

- declaration: error ;
  It tells the parser that when there is an error, it should ignore the token and all the following tokens until it finds the next semicolon.

Symbol Table and Constants Table

The symbol table displays the functions and identifiers. The constants table displays the constants along with their type, i.e., String Literal, Float, Integer, etc. Hash table has been used to implement symbol and constants table. The constants table and symbol table don’t display duplicates.

Output of Parser

If we wish to abort the program after the first syntax error, then we need to include the following statement in yyerror() function:

```
exit(1);
```

If we wish to display all the syntax errors (along with line numbers) in the program, the message displayed by parser is “Parsing Complete” irrespective of whether there are syntax errors in program or not. This is because the function yyparse() returns the value 0.

The function yyparse() returns 0 when it reaches the end of the input file while parsing.
It returns 1 if parsing failed because of invalid input. (YYABORT is invoked)
It returns 2 if parsing failed due to memory exhaustion.

Our parser currently has no shift-reduce conflicts. We have mentioned how shift-reduce conflicts which initially occurred were handled by us in the later section.

4. TEST CASES

Test Case #1

Input

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

int main()
{
int a=6, b;
while ( a < b; )
{
     a += 1;
     b -= 1;
}
if ( a<10 )
     printf("Value is less than 10");
else
{
     if ( a>17 )
         printf("Value is greater than 17");
     else
         printf("Value is lesser than 17");
     else
         printf("Error in calculation");
}
}
```

Output

```
Line 7 : syntax error

Line 20 : syntax error

__________CONSTANTS TABLE__________

Name                            Type
6                               Integer
1                               Integer
10                              Integer
"Value is less than 10"                                 String Literal
17                              Integer
"Value is greater than 17"                              String Literal
"Value is lesser than 17"                               String Literal
"Error in calculation"                          String Literal


__________SYMBOLS TABLE___________

Name                            Type
main                            Function
a                               Identifier
b                               Identifier
printf                          Function



Parsing complete
```

Test- Case #2

Input

```
#include<stdio.h>

union u
{
  int a,b;
}boat;

struct s
{
  int a, b;
}car

int main()
{
  printf("%d", car.a);
  printf("%d", boat->.a);
  int value=6
  printf("%d",value)
}
```

Output

```
Line 14 : syntax error

Line 16 : syntax error

Line 18 : syntax error

Line 19 : syntax error

__________CONSTANTS TABLE__________

Name                            Type
"%d"                            String Literal
6                               Integer


__________SYMBOLS TABLE__________

Name                            Type
u                               Identifier
a                               Identifier
b                               Identifier
boat                            Identifier
s                               Identifier
car                             Identifier
main                            Function
printf                          Function
value                           Identifier



Parsing complete
```

Test- Case #3

Input

```
#include<stdio.h>
#define b 7

int main(int argc. int *argv[])
{
  int a=9, c=3;
  switch(a)
  {
     case 1: switch(c)
        {
           case 2: printf("%d",i);
           break;
           case 3: printf("Value of c is 4");
        }
          break;
     case 6 printf("Value of a is 6");
     default: printf("None of the values");
  }
}
```

Output

```
Line 4 : syntax error

Line 16 : syntax error

_________CONSTANTS TABLE__________

Name                        Type
9                           Integer
3                           Integer
1                           Integer
2                           Integer
"%d"                        String Literal
"Value of c is 4"                           String Literal
6                           Integer
"Value of a is 6"                           String Literal
"None of the values"                        String Literal


_________SYMBOLS TABLE__________

Name                        Type
main                        Function
argc                        Identifier
argv                        Identifier
a                           Identifier
c                           Identifier
printf                      Function
i                           Identifier



Parsing complete
```

Test-Case #4

Input

```
#include<stdio.h>

int main()
{
  int i, a=4;
  for(i=0;i<10)
    printf("Hi");
  do
    {
       printf("%d",a);
       j=3;
       a++;
       do;
       {
         printf("Bye");
         j++;
       }while(j<=10);
    }while(a<10);
}
```

Output

```
Line 6 : syntax error

Line 14 : syntax error

__________CONSTANTS TABLE__________

Name                          Type
4                             Integer
0                             Integer
10                            Integer
"Hi"                          String Literal
"%d"                          String Literal
3                             Integer
"Bye"                         String Literal


__________SYMBOLS TABLE__________

Name                          Type
main                          Function
i                             Identifier
a                             Identifier
printf                        Function
j                             Identifier



Parsing complete
```

Test-Case #5

Input

```
#include<stdio.h>

struct st
{
int a,b;
}example;

int main()
{
int a=1,b=3;

// We try to compare the values of a and b

if (a<b)
  printf("Value of b is more");
else
  {
    if (a==b)
      printf("a and b are equal");
    else
      printf("Value of a is more");
  }
printf("%d\n",example.a);

}
```

Output

```
__________CONSTANTS TABLE__________

Name                            Type
1                               Integer
3                               Integer
"Value of b is more"                            String Literal
"a and b are equal"                             String Literal
"Value of a is more"                            String Literal
"%d\n"                          String Literal


__________SYMBOLS TABLE__________

Name                            Type
st                              Identifier
a                               Identifier
b                               Identifier
example                                 Identifier
main                            Function
printf                          Function



Parsing complete
```

5. PRODUCTION RULES

The production rules for many features of C are fairly straightforward. Production rules for the important features of C are as follows:

- **Labelled Statement**

  ```
  labeled_statement
  : IDENTIFIER ':' statement
  | CASE constant_expression ':' statement
  | DEFAULT ':' statement
  ;
  ```

- **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
  | DO statement WHILE '(' expression ')' ';'
  | FOR '(' expression_statement expression_statement ')' statement
  | FOR '(' expression_statement expression_statement expression ')' statement
  ;
  ```

- **Jump Statement**

  ```
  jump_statement
  : GOTO IDENTIFIER ';'
  | CONTINUE ';'
  | BREAK ';'
  | RETURN ';'
  | RETURN expression ';'
  ;
  ```

Results

The lex (scanner.l) and yacc (parser.y) codes are compiled and executed by the following terminal commands to parse the given input file (testcase_1.txt):

```
lex scanner.l
yacc parser.y
gcc y.tab.c -ll -ly
./a.out  testcase_1.txt
```

After parsing, if there are errors then the line numbers of those errors are displayed. Otherwise, a 'Parsing Complete' message is displayed on the screen.

6. HANDLING SHIFT-REDUCE CONFLICTS

Introduction:

The parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a shift / reduce conflict. It may also happen that the parser has a choice of two legal reductions; this is called a reduce / reduce conflict. Note that there are never any ``shift/shift'' conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguating rule.
Yacc invokes the following disambiguating rules by default:

- In a shift/reduce conflict, the default is to do the shift.
- Rule 1 implies that reductions are deferred whenever there is a choice, in favour of shifts.

Reduce/reduce conflicts should be avoided whenever possible.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors.

Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parser even in the presence of conflicts.

%token does not set the precedence of a token -- it declares the token to exist with NO precedence. If precedence has to be declared for a token, %left, %right or %nonassoc has to be used -- all of which both declare the token AND set its precedence.

Solving dangling else problem:

There is a much simpler solution. In an LR parser, 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 to 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 the rule (e.g., NO_ELSE), and it is specified 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
;
```

Possibilities of Shift/reduce conflicts and handling them:

Dangling Else Problem

Suppose we are parsing a language which has if-then and if-then-else statements, with a pair of rules like this:

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

Here IF and ELSE are terminal symbols for specific keyword tokens.
When the ELSE token is read and becomes the lookahead token, the contents of the stack are just right for reduction by the first rule. But it is also legitimate to shift the ELSE, because that would lead to eventual reduction by the second rule.

The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal.

This situation, where either a shift or a reduction would be valid, is called a *shift/reduce conflict*. Bison is designed to resolve these conflicts by choosing to shift, unless otherwise directed by operator precedence declarations.

The conflict exists because the grammar as written is ambiguous: either parsing of the simple nested if-statement is legitimate. The established convention is that these ambiguities are resolved by attaching the else-clause to the innermost if-statement; this is what Bison accomplishes by choosing to shift rather than reduce. This particular ambiguity is called the “dangling else” ambiguity.

Precedence and Associativity of Operators

One common source of shift/reduce conflicts is using an ambiguous grammar for expressions that does not specify the associativities and precedence levels of its operators.
Example: Grammar which does not specify the associativities of the + and * operators or their relative precedence.

Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form
expr : expr OP expr
and
expr : UNARY expr

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. Thus,
%left '+' '-'
%left '*' '/'
describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

- The precedences and associativities are recorded for those tokens and literals that have them.

- A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

- If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Shift/reduce conflicts are harmless as they are anyways handled by the Yacc.
However, in a truely ambiguous grammar, doing the shift will send the parser down one of two paths, only one of which will ultimately find a string in the grammar. In this case the S/R conflict is a fatal error.

Precedences:

Yacc allows specifying operator precedences with %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 defining only precedence and no associativity at all.

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 the 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.

7. PARSE TREE

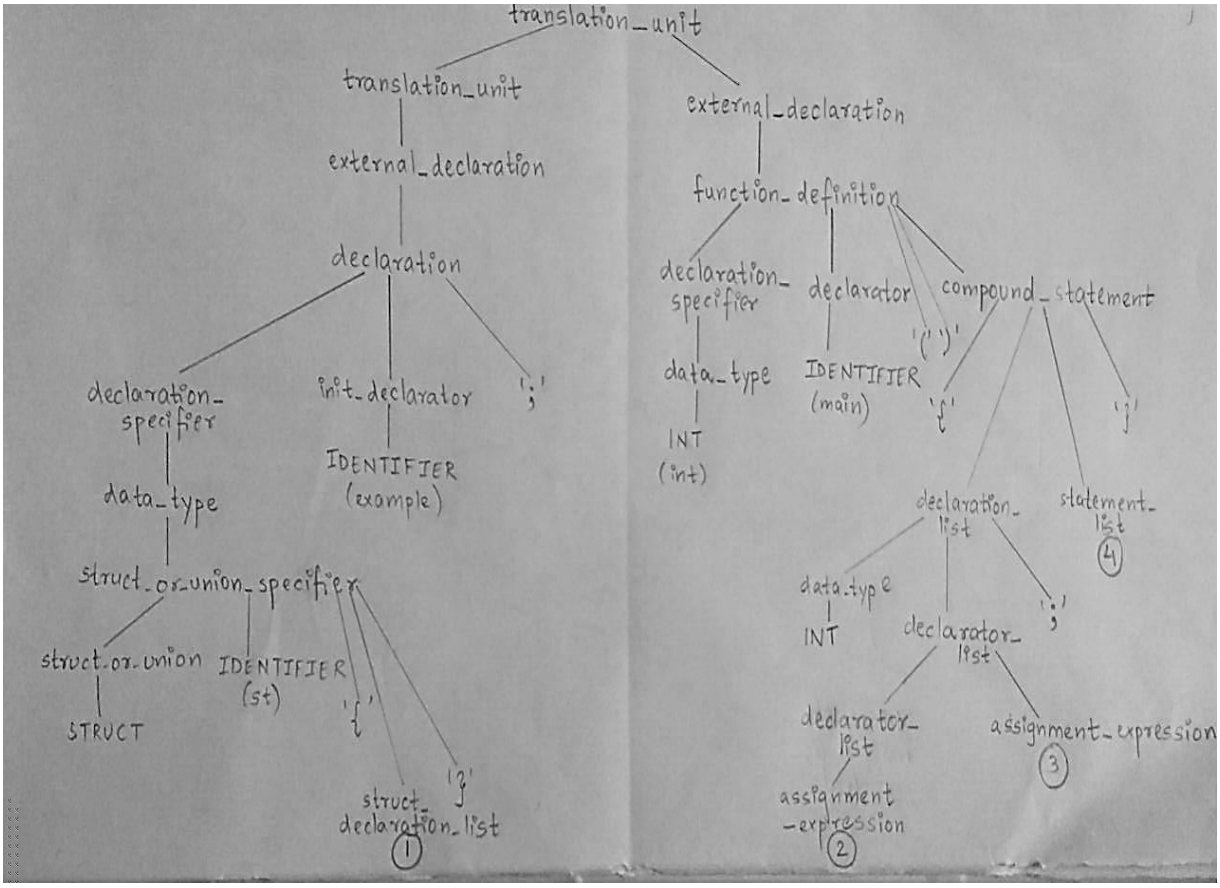
A parse tree has been drawn for Test-Case #5 (which has no syntax errors).

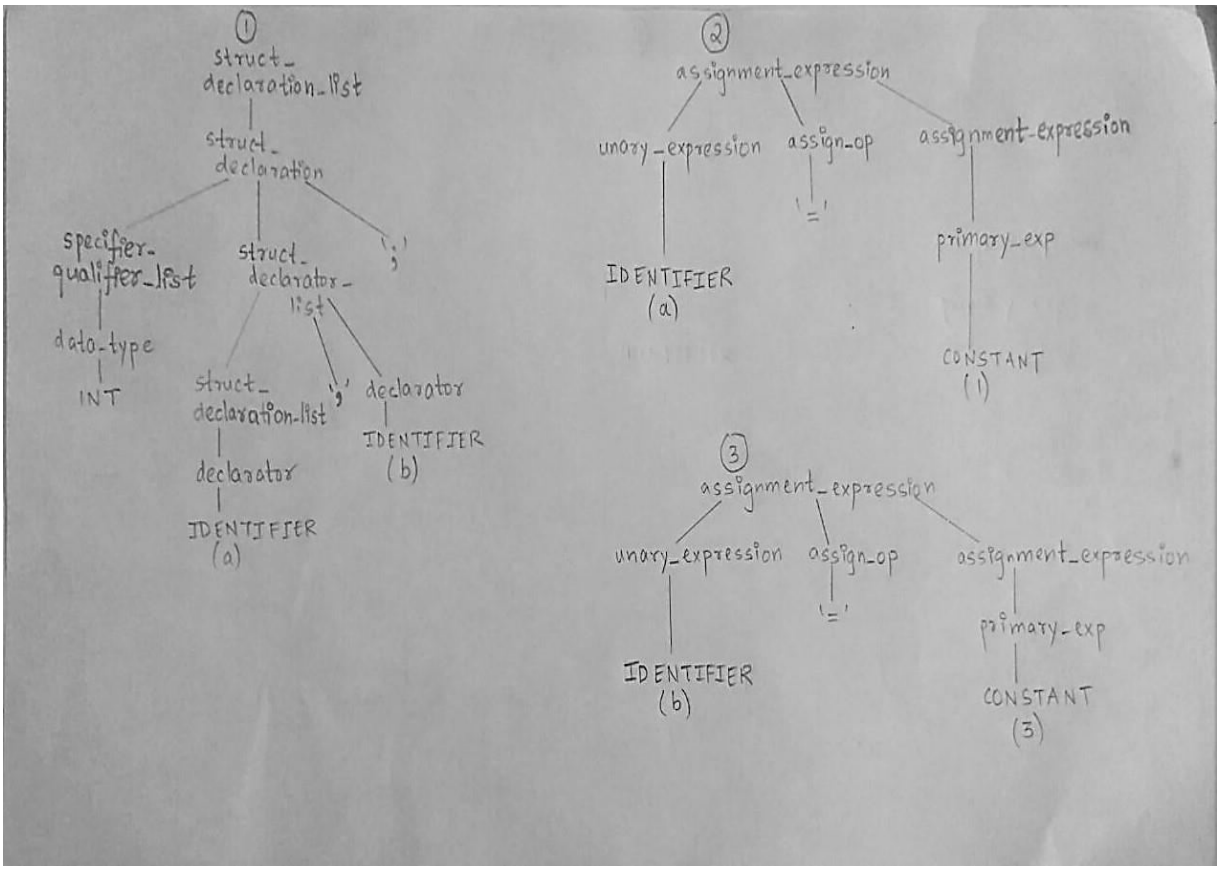
In the parse tree and the production rules, the non-terminal expression_statement is used for function calling.

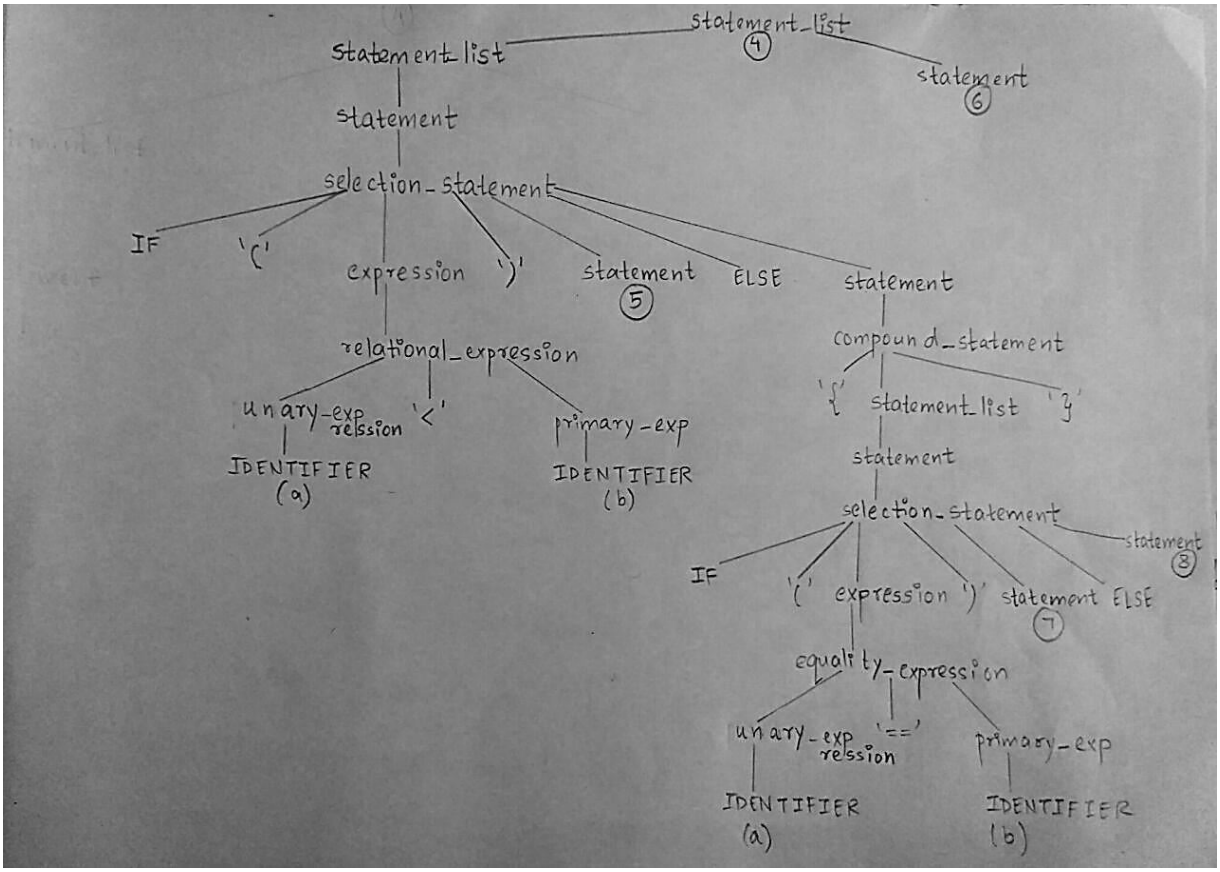
A parse tree for a grammar G is a tree where

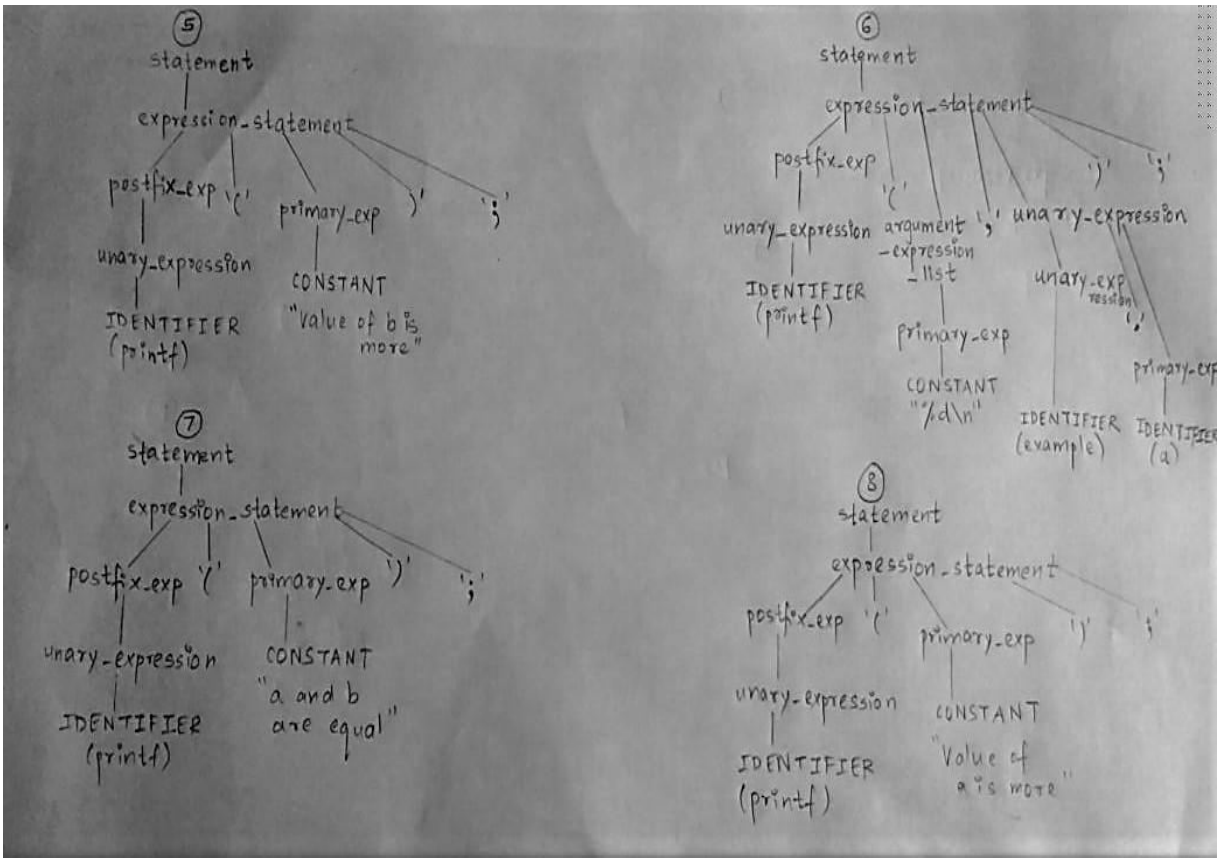
- the root is the start symbol for G
- the interior nodes are the nonterminals of G
- the leaf nodes are the terminal symbols of G.
- the children of a node T (from left to right) correspond to the symbols on the right hand side of some production for T in G.

Every terminal string generated by a grammar has a corresponding parse tree; every valid parse tree represents a string generated by the grammar (called the yield of the parse tree).


translation_unit
translation_unit
external_declaration
declaration
declaration-specifier
data_type
struct_or_union_specifier
struct_or_union
STRUCT
IDENTIFIER (st)
'{'
'}'
struct-declaration_list
(1)
init-declarator
IDENTIFIER (example)
';'
external_declaration
function_definition
declaration-specifier
data_type
INT (int)
declarator
IDENTIFIER (main)
'('
')'
compound_statement
'{'
'}'
declaration_list
statement_list
(4)
data_type
INT
declarator_list
';'
declarator_list
assignment-expression
(2)
assignment_expression
(3)


(1)
struct_declaration_list
struct_declaration
specifier-qualifier_list
data-type
INT
struct_declarator_list
';'
struct_declaration-list
declarator
IDENTIFIER (a)
';'
declarator
IDENTIFIER (b)
(2)
assignment_expression
unary_expression
IDENTIFIER (a)
assign_op
'='
assignment_expression
primary_exp
CONSTANT (1)
(3)
assignment_expression
unary_expression
IDENTIFIER (b)
assign_op
'='
assignment_expression
primary_exp
CONSTANT (3)


statement_list
statement_list
4
statement
6
statement
selection_statement
IF
'('
expression
')'
statement
5
ELSE
statement
relational_expression
unary-expression
'<'
primary-exp
IDENTIFIER
(a)
IDENTIFIER
(b)
compound_statement
'{'
statement_list
'}'
statement
selection_statement
statement
8
IF
'('
expression
')'
statement
7
ELSE
equality_expression
unary-expression
'=='
primary-exp
IDENTIFIER
(a)
IDENTIFIER
(b)


5
statement
expression_statement
postfix_exp
'('
primary_exp
')'
';'
unary_expression
CONSTANT
"Value of b is more"
IDENTIFIER
(printf)
7
statement
expression_statement
postfix_exp
'('
primary_exp
')'
';'
unary_expression
CONSTANT
"a and b are equal"
IDENTIFIER
(printf)
6
statement
expression_statement
postfix_exp
'('
argument_expression_list
','
unary_expression
')'
';'
unary_expression
IDENTIFIER
(printf)
primary_exp
CONSTANT
"%d\n"
unary-expression
','
primary-exp
IDENTIFIER
(example)
IDENTIFIER
(a)
8
statement
expression_statement
postfix_exp
'('
primary_exp
')'
';'
unary_expression
CONSTANT
"Value of a is more"
IDENTIFIER
(printf)

8. FUTURE WORK

The result of the project will be a lex file and a yacc file which act as a parser for C language. A *parser* takes input in the form of a sequence of tokens or program instructions and builds a data structure in the form of a parse tree or an abstract syntax tree.

We have removed the shift-reduce conflicts pertaining to precedence and associativity in our code. In the future, we will try implementing more pre-defined C functions and multiple function handling.

We will also develop a semantic analyzer and an intermediate code generator for C language and make sure that all the components of the project work with each other to develop a good compiler.

9. REFERENCES

- Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
- Lex and Yacc By John R. Levine, Tony Mason, Doug Brown
- epaperpress.com/lexandyacc
- cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf