

Compiler Design Project Submission - 2

Topic

Parser Code for C language

Submitted by

15CO239 – R.RuhiTaj

15CO253 – Y. M. Greeshma

Date of Submission

February 20, 2018

Content:

- Introduction to Parser
- Yacc code
- Lex code
- help.h
- How to compile the programs
- Test cases, output with screen shots
- Parse tree
- Conclusion

Compiler Design Project Submission - 2

Introduction to Parsing

What is PARSING?

Parsing , syntax analysis or syntactic analysis is the process of analysing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. The term *parsing* comes from Latin *pars (orationis)*, meaning part (of speech).

The term has slightly different meanings in different branches of linguistics and computer science. Traditional sentence parsing is often performed as a method of understanding the exact meaning of a sentence or word, sometimes with the aid of devices such as sentence diagrams. It usually emphasizes the importance of grammatical divisions such as subject and predicate.

What is a PARSER?

A **parser** is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure – giving a structural representation of the input, checking for correct syntax in the process.

The parsing may be preceded or followed by other steps, or these may be combined into a single step. The parser is often preceded by a separate lexical analyser, which creates tokens from the sequence of input characters; alternatively, these can be combined in scannerless parsing.

Parsers may be programmed by hand or may be automatically or semi-automatically generated by a parser generator. Parsing is complementary to templating, which produces formatted output. These may be applied to different domains, but often appear together, such as the scanf/printf pair, or the input (front end parsing) and output (back end code generation) stages of a compiler.

The input to a parser is often text in some computer language, but may also be text in a natural language or less structured textual data, in which case generally only certain parts of the text are extracted, rather than a parse tree being constructed.

Parsers range from very simple functions such as scanf, to complex programs such as the frontend of a C++ compiler or the HTML parser of a web browser. An important class of simple parsing is done using regular expressions, in which a group of regular expressions defines a regular language and a regular expression engine automatically generating a parser for that language, allowing pattern matching and extraction of text. In other contexts regular expressions are instead used prior to parsing, as the lexing step whose output is then used by the parser.

Compiler Design Project Submission - 2

Role of PARSER in compilation:

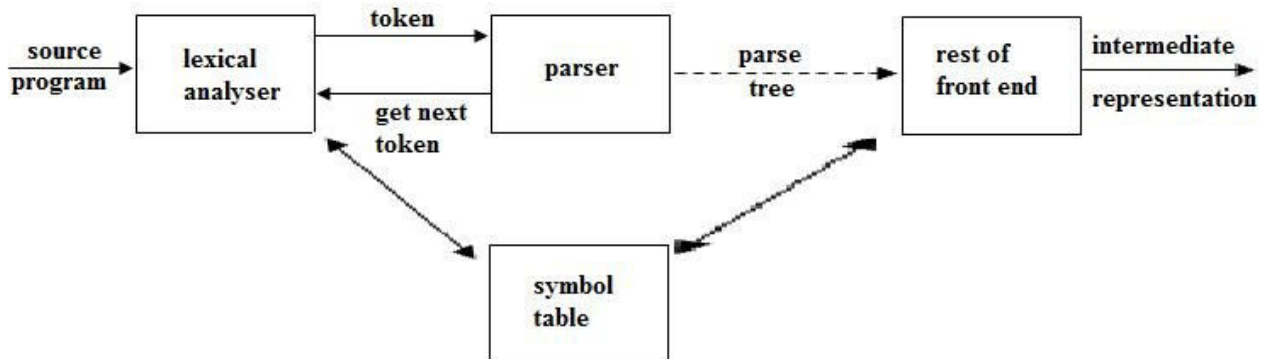
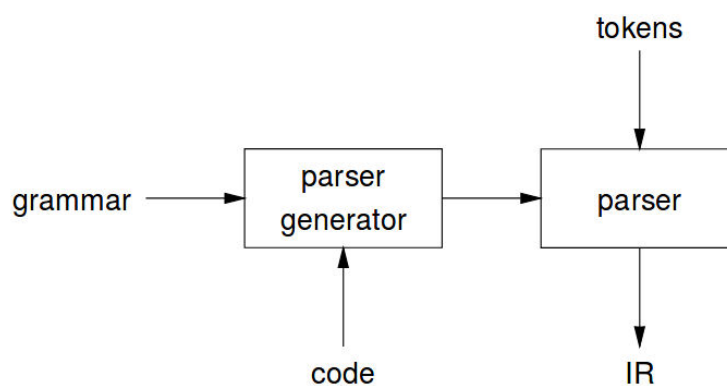


Fig 2.1 Position of parser in compiler model

Functions of PARSER in compilation:

- Performs context-free syntax analysis
- Guides context-sensitive analysis
- Constructs an intermediate representation
- Produces meaningful error messages
- Attempts error correction

Parsing: the big picture



Types of parsers

Compiler Design Project Submission - 2

The *task* of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

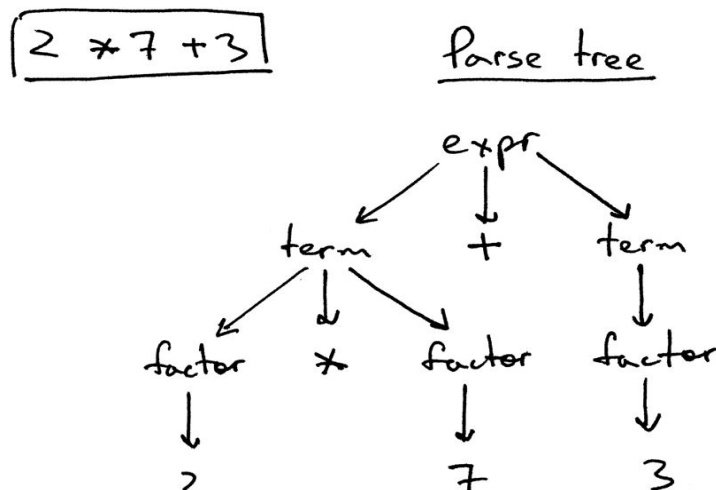
- **Top-down parsing** - Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.[5]
- **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.

Parse Tree:

A parse tree or parsing tree[1] or derivation tree or concrete syntax tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar. The term *parse tree* itself is used primarily in computational linguistics; in theoretical syntax, the term *syntax tree* is more common.

Parse trees concretely reflect the syntax of the input language, making them distinct from the abstract syntax trees used in computer programming. Unlike Reed-Kellogg sentence diagrams used for teaching grammar, parse trees do not use distinct symbol shapes for different types of constituents.

Example of Simple Parse Tree for the simple Expression:



Implementation of Parser using yacc

Compiler Design Project Submission - 2

Grammar definitions are used to define a sequence of tokens and what the result should be. In general this is used in combination with lex, which actually reads the text, and the two make up the parser.

Grammar sequence and precedence

In most languages, whether textual, mathematical, or programming, there is usually some sort of precedence or significance to different phrases that give them priority over similar, but different, combinations.

For example, in most programming languages within a mathematical expression, multiplication has higher precedence than addition or subtraction. For example, the expression: $4+5*6$ evaluates to: $4+30$.

This ultimately evaluates to 34. The reason is that the multiplication operation happens first (5 times 6 equals 30) and this is then used in the final calculation, which adds the result to 4 to give 34.

Within yacc, you define the precedence by defining multiple grammar sequence groups and linking them; the order of the individual expressions within the group helps to define the precedence.

Exchanging data with lex

The primary method of data exchange between lex and yacc is the token definitions that are generated when you create the C source from yacc and generate the header file that contains the C macros for each token.

In order to understand the information supplied to a text parsing application, there are two phases. The first is simply to identify what has been typed or provided to an application. You must be able to identify the key words, phrases, or character sequences from the input source so that you can determine what to do with them. The second process is to understand the structure of that information -- the grammar -- so that the input can be both validated and operated on. An excellent example of grammar is the use of parentheses in most programming languages.

Output

With the combination of **lex** and **yacc**, we can generate the code that builds a parser. The lex tool provides a method for identifying text fragments and elements in order to return tokens. The yacc tool provides a method of taking the structure of those tokens using a series of rules to describe the format of the input and defines a method for dealing with the identified sequences. In both cases, the tools use a configuration file, which when processed generates the C source code to build the appropriate parsing application.

Yacc File Format:

Compiler Design Project Submission - 2

```
%{  
/* Global and header definitions required */  
%}  
  
/* Declarations (Optional token definitions) */  
  
%%  
/* Parsing ruleset definitions  
%%  
  
/* Additional C source code */
```

Code for Lex :

The code below is the lex code which contains the definitions of tokens and tokenisation:

project.l – .l is the extension for a lex program

Compilation of project.l:

Command in Ubuntu : `lex project.l`

*The compilation of this lex file creates a **lex.yy.c** file!*

CODE

```
%option yylineno  
  
%{  
    #include<stdio.h>  
    #include"y.tab.h"  
    #include<math.h>  
    #include "help.h"  
  
%}  
  
space[ ]  
openArrow[<]  
onlyAlphabets[a-zA-Z]  
closeArrow[>]  
dot[.]  
  
%%
```

Compiler Design Project Submission - 2

```
"#include"([ ]+)?((<(\.|\^[>])+>)|(\\"(\.|\^[")]+\"))
{ the_medium_to_display(yytext, 1);return HEADER;}
```

```
"#define"[ ]+[a-zA-z][a-zA-z_0-9]*
{the_medium_to_display(yytext, 2); return DEFINE; }
```

```
"auto"|"register"|"static"|"extern"|"typedef"
{ the_medium_to_display(yytext, 3); return storage_const;}
```

```
"void"|"char"|"short"|"int"|"long"|"float"|"double"|"signed"|"unsigned"
    { the_medium_to_display(yytext, 4); return type_const;}
```

```
"const"|"volatile" { the_medium_to_display(yytext, 5); return qual_const;}
```

```
"enum"                {the_medium_to_display(yytext, 6); return enum_const; }
```

```
"case"                { the_medium_to_display(yytext, 7); return CASE;}
```

```
"default"            { the_medium_to_display(yytext, 8); return DEFAULT;}
```

```
"if"      { the_medium_to_display(yytext, 9); return IF;}
```

```
"switch" {the_medium_to_display(yytext, 10); return SWITCH; }
```

```
"else" { the_medium_to_display(yytext, 11);return ELSE;}
```

```
"for"  { the_medium_to_display(yytext, 12); return FOR;}
```

```
"do" { the_medium_to_display(yytext, 13); return DO;}
```

```
"while"      { the_medium_to_display(yytext, 14); return WHILE;}
```

Compiler Design Project Submission - 2

```
"goto"      { the_medium_to_display(yytext, 15); return GOTO;}

"continue"   {the_medium_to_display(yytext, 16); return CONTINUE; }

"break"      { the_medium_to_display(yytext, 17); return BREAK;}

"struct"|"union"  { the_medium_to_display(yytext, 18); return struct_const;}

"return"     {the_medium_to_display(yytext, 19); return RETURN; }

"sizeof"     { the_medium_to_display(yytext, 20); return SIZEOF;}

"|" {the_medium_to_display(yytext, 21); return or_const; }

"&&" {the_medium_to_display(yytext, 22); return and_const; }

"..." { the_medium_to_display(yytext, 23); return param_const;}

"=="|"!=" { the_medium_to_display(yytext, 24); return eq_const;}

"<="|">=" {the_medium_to_display(yytext, 25); return rel_const; }

">>"|"<<" { the_medium_to_display(yytext, 26); return shift_const;}

"++"|"--" { the_medium_to_display(yytext, 27); return inc_const;}

"->" {the_medium_to_display(yytext, 28); return point_const; }

";"|"="|"{"|"}|"(")|")|"["|"]|"*"|"+"|"-"|" "/"|"?"|":"|"&"|"|"|"^"|"!"|"~"|"%"|"<"|">"

{ the_medium_to_display(yytext, 29); return yytext[0];}
```


Compiler Design Project Submission - 2

"*="|"/="|"+="|"%"="|">="|"-="|"<="|"&="|"^="|'|="

{the_medium_to_display(yytext, 30); return PUNC; }

[0-9]+ { the_medium_to_display(yytext, 31); return int_const;}

[0-9]+ "." [0-9]+ { the_medium_to_display(yytext, 32); return float_const; }

""'. "" "" "" [a-zA-z_][a-zA-z_0-9]* "" { the_medium_to_display(yytext, 33); return char_const ;}

[a-zA-z_][a-zA-z_0-9]* { the_medium_to_display(yytext, 34); return id;}

\"(\\.|[^\"])*\" { the_medium_to_display(yytext, 35); return string;}

"/" (\\.|[^\n])*[\\n] ;

[/][*]([^[*])[[*]*[^*/])*[*]+[/] ;

[\\t\\n] ;

%%

int yywrap(void)

```
{
    return 1;
}
```

Compiler Design Project Submission - 2

Code for Yacc :

The code below is yacc code which contains the parsing of tokens and rules of grammar for the creation of parse trees:

project.y – .y is the extension for a yacc program

Compilation of project.y:

Command in Ubuntu : `yacc -v -d project.y`

*This command creates **three** separate files.*

- *-v helps in creation of y.tab.c*
- *-d helps in creation of y.tab.h*
- *The complete compilation creates a y.output*

Then, if these three files are created, the compilation is successful!

CODE

```
%{
```

```
    #include<stdio.h>
    # include <string.h>
    # include <stdlib.h>
    int yylex(void);
    int yyerror(const char *s);
    int success = 1;
```

```
%}
```

```
%token int_const char_const float_const id string enumeration_const storage_const type_const
qual_const struct_const enum_const DEFINE
```

Compiler Design Project Submission - 2

```
%token IF FOR DO WHILE BREAK SWITCH CONTINUE RETURN CASE DEFAULT GOTO
sizeof PUNC or_const and_const eq_const shift_const rel_const inc_const
```

```
%token point_const param_const ELSE HEADER
```

 $\%left '+' '-'$ $\%left '*' '/'$

```
%nonassoc "then"
```

```
%nonassoc ELSE
```

```
%define parse.error verbose
```

```
%start program_unit
```

%%

```

program_unit      :      HEADER program_unit
                   | DEFINE primary_exp program_unit
                   | translation_unit

```

;

```
translation_unit      : external_decl
                        | translation_unit external_decl
```

;

Compiler Design Project Submission - 2

```
external_decl          : function_definition
                        | decl
                        ;

function_definition    : decl_specs declarator decl_list compound_stat
                        | declarator decl_list compound_stat
                        | decl_specs declarator      compound_stat
                        | declarator compound_stat
                        ;

decl                   : decl_specs init_declarator_list ';'
                        | decl_specs ';'
                        ;

decl_list              : decl
                        | decl_list decl
                        ;

decl_specs             : storage_class_spec decl_specs
                        | storage_class_spec
                        | type_spec decl_specs
                        | type_spec
                        | type_qualifier decl_specs
                        | type_qualifier
                        ;

storage_class_spec     : storage_const
                        ;

type_spec              : type_const
                        | struct_or_union_spec
                        | enum_spec
                        | typedef_name
                        ;
```

Compiler Design Project Submission - 2

```
type_qualifier          : qual_const
                        ;

struct_or_union_spec    : struct_or_union id '{' struct_decl_list '}'
                        | struct_or_union '{' struct_decl_list '}'
                        | struct_or_union id
                        ;

struct_or_union         : struct_const
                        ;

struct_decl_list        : struct_decl
                        | struct_decl_list struct_decl
                        ;

init_declarator_list    : init_declarator
                        | init_declarator_list ',' init_declarator
                        ;

init_declarator         : declarator
                        | declarator '=' initializer
                        ;

struct_decl             : spec_qualifier_list struct_declarator_list ';'
                        ;

spec_qualifier_list     : type_spec spec_qualifier_list
                        | type_spec
                        | type_qualifier spec_qualifier_list
                        | type_qualifier
                        ;

struct_declarator_list  : struct_declarator
                        | struct_declarator_list ',' struct_declarator
                        ;

struct_declarator       : declarator
                        | declarator ':' const_exp
                        | ':' const_exp
                        ;
```

Compiler Design Project Submission - 2

```
enum_spec          : enum_const id '{' enumerator_list '}'
                   | enum_const '{' enumerator_list '}'
                   | enum_const id
                   ;

enumerator_list    : enumerator
                   | enumerator_list ',' enumerator
                   ;

enumerator         : id
                   | id '=' const_exp
                   ;

declarator         : pointer direct_declarator
                   | direct_declarator
                   ;

direct_declarator  : id

                   | '(' declarator ')'

                   | direct_declarator '[' const_exp ']'

                   | direct_declarator '[' ']'

                   | direct_declarator '(' param_type_list ')'

                   | direct_declarator '(' id_list ')'

                   | direct_declarator '(' ' ' ')'

                   ;

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

Compiler Design Project Submission - 2

```
type_qualifier_list      : type_qualifier
                          | type_qualifier_list type_qualifier
                          ;

param_type_list          : param_list
                          | param_list ',' param_const
                          ;

param_list               : param_decl
                          | param_list ',' param_decl
                          ;

param_decl               : decl_specs declarator
                          | decl_specs abstract_declarator
                          | decl_specs
                          ;

id_list                  : id
                          | id_list ',' id
                          ;

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

initializer_list          : initializer
                          | initializer_list ',' initializer
                          ;

type_name                 : spec_qualifier_list abstract_declarator
                          | spec_qualifier_list
                          ;

abstract_declarator       : pointer
                          | pointer direct_abstract_declarator
                          |      direct_abstract_declarator
                          ;

direct_abstract_declarator : '(' abstract_declarator ')'
                          | direct_abstract_declarator '[' const_exp ']'
```

Compiler Design Project Submission - 2

```

| '[' const_exp ']'
| direct_abstract_declarator '[' ']'
| '[' ']'
| direct_abstract_declarator '(' param_type_list ')'
| '(' param_type_list ')'
| direct_abstract_declarator '(' ')'
| '(' ')'
;

typedef_name      : 't'
                  ;

stat              : labeled_stat

                  | exp_stat

                  | compound_stat

                  | selection_stat

                  | iteration_stat
                  | jump_stat
                  ;

labeled_stat      : id ':' stat

                  | CASE const_exp ':' stat
                  | DEFAULT ':' stat
                  ;

exp_stat          : exp ';'

                  | ';'
                  ;

compound_stat     : '{' decl_list stat_list '}'

                  | '{' stat_list '}'
```


Compiler Design Project Submission - 2

```

| '{' decl_list  '}'

| '{' '}'

;

stat_list          : stat

| stat_list stat

;

selection_stat     : IF '(' exp ')' stat
                    %prec "then"
                    | IF '(' exp ')' stat ELSE stat
                    | SWITCH '(' exp ')' stat
                    ;

iteration_stat      : WHILE '(' exp ')' stat
                    | DO stat WHILE '(' exp ')' ';'
                    | FOR '(' exp ';' exp ';' exp ')' stat
                    | FOR '(' exp ';' exp ';' ')' stat
                    | FOR '(' exp ';' ';' exp ')' stat
                    | FOR '(' exp ';' ';' ')' stat
                    | FOR '(' ';' exp ';' exp ')' stat
                    | FOR '(' ';' exp ';' ')' stat
                    | FOR '(' ';' ';' exp ')' stat
                    | FOR '(' ';' ';' ')' stat
                    ;

jump_stat          : GOTO id ';'
                    | CONTINUE ';'
                    | BREAK ';'
                    | RETURN exp ';'
                    | RETURN ';'
                    ;
```

Compiler Design Project Submission - 2

```
exp                                     : assignment_exp
                                     | exp ',' assignment_exp
                                     ;

assignment_exp                         : conditional_exp
                                     | unary_exp assignment_operator
assignment_exp
                                     ;

assignment_operator                   : PUNC
                                     | '='
                                     ;

conditional_exp                       : logical_or_exp
                                     | logical_or_exp '?' exp ':' conditional_exp
                                     ;

const_exp                            : conditional_exp
                                     ;

logical_or_exp                       : logical_and_exp
                                     | logical_or_exp or_const logical_and_exp
                                     ;

logical_and_exp                      : inclusive_or_exp
                                     | logical_and_exp and_const inclusive_or_exp
                                     ;

inclusive_or_exp                     : exclusive_or_exp
                                     | inclusive_or_exp '|' exclusive_or_exp
                                     ;

exclusive_or_exp                     : and_exp
                                     | exclusive_or_exp '^' and_exp
                                     ;

and_exp                              : equality_exp
                                     | and_exp '&' equality_exp
                                     ;
```

Compiler Design Project Submission - 2

```
equality_exp          : relational_exp
                        | equality_exp eq_const relational_exp
                        ;

relational_exp        : shift_expression
                        | relational_exp '<' shift_expression
                        | relational_exp '>' shift_expression
                        | relational_exp rel_const shift_expression
                        ;

shift_expression      : additive_exp
                        | shift_expression shift_const additive_exp
                        ;

additive_exp          : mult_exp
                        | additive_exp '+' mult_exp
                        | additive_exp '-' mult_exp
                        ;

mult_exp              : cast_exp
                        | mult_exp '*' cast_exp
                        | mult_exp '/' cast_exp
                        | mult_exp '%' cast_exp
                        ;

cast_exp              : unary_exp
                        | '(' type_name ')' cast_exp
                        ;

unary_exp             : postfix_exp
                        | inc_const unary_exp
                        | unary_operator cast_exp
                        | SIZEOF unary_exp
                        | SIZEOF '(' type_name ')'
                        ;

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

Compiler Design Project Submission - 2

```
postfix_exp                : primary_exp

                             | postfix_exp '[' exp ']'
                             | postfix_exp '(' argument_exp_list ')'
                             | postfix_exp '(' ')'
                             | postfix_exp '.' id
                             | postfix_exp point_const id
                             | postfix_exp inc_const
                             ;

primary_exp                 : id

                             | consts

                             | string

                             | '(' exp ')'
                             ;

argument_exp_list           : assignment_exp
                             | argument_exp_list ',' assignment_exp
                             ;

consts                      : int_const

                             | char_const
                             | float_const
                             | enumeration_const
                             ;

%%
```

Compiler Design Project Submission - 2

```
int yyerror(const char *msg)
{
    extern int yylineno;
    printf("There is an error and failed to parse.\nLine Number: %d %s\n",yylineno,msg);
    success = 0;
    return 0;
}

int main()
{
    yyparse();
    if(success)
    {
        printf("Parsing Successful\n");

    }
    theTable();

    return 0;
}
```

Compiler Design Project Submission - 2

Code for help.h

This file contains the information about the tokens.

The method of storage is all included in this file help.h and is include in the lex file, where tokenisation of C file is done.

CODE

```
char* everything[36][50];
int i,count[36] = {0};

void namethe(int no){

switch(no){

case 1: printf("\n\n| Header_File      ");
        break;
case 2: printf("\n\n| Define      ");
        break;
case 3: printf("\n\n| Storage_Constant      ");
        break;
case 4:   printf("\n\n| Type_Constant      ");
        break;
case 5: printf("\n\n| Qual_Constant      ");
        break;
case 6: printf("\n\n| Enum_Constant      ");
        break;
case 7: printf("\n\n| Case      ");
        break;
```

Compiler Design Project Submission - 2

```
case 8: printf("\n\n| Default      ");
        break;
case 9: printf("\n\n| If   ");
        break;
case 10: printf("\n\n| Switch      ");
        break;
case 11: printf("\n\n| Else   ");
        break;
case 12: printf("\n\n| For    ");
        break;
case 13: printf("\n\n| Do     ");
        break;
case 14: printf("\n\n| While   ");
        break;

case 15: printf("\n\n| Goto    ");
        break;

case 16: printf("\n\n| Continue  ");
        break;

case 17: printf("\n\n| Break    ");
        break;

case 18: printf("\n\n| Struct_Constant  ");
        break;

case 19: printf("\n\n| Return   ");
        break;
```

Compiler Design Project Submission - 2

```
case 20:printf("\n\n| Size_Of   ");
        break;

case 21:printf("\n\n| Or_Const   ");
        break;
case 22:printf("\n\n| And_Const  ");
        break;
case 23:printf("\n\n| Param_Const ");
        break;
case 24:printf("\n\n| Equi_Const ");
        break;
case 25:printf("\n\n| Rel_Const  ");
        break;
case 26:printf("\n\n| Shift_Const ");
        break;
case 27:printf("\n\n| Inc_Const  ");
        break;
case 28:printf("\n\n| Point_Const ");
        break;


case 29:printf("\n\n| Just_Symbol ");
        break;


case 30:printf("\n\n| Punc     ");
        break;
case 31:printf("\n\n| int_Const  ");
        break;
case 32:printf("\n\n| float_Const ");
        break;
```


Compiler Design Project Submission - 2

```
case 33:printf("\n\n| Char_Const   ");
        break;
case 34:printf("\n\n| id      ");
        break;
case 35:printf("\n\n| string   ");
        break;

default: break;
}

}

void the_medium_to_display(char* word, int no){

if(no==1){

the_medium_to_display(word,35);

}

int y = strlen(word);

everything[no][count[no]] = (char*)malloc(y*sizeof(char));
int g;

for(g=0;g<y;g++){

everything[no][count[no]][g] = word[g];
}
```

Compiler Design Project Submission - 2

```
count[no] = count[no]+1;
//namethe(no);

//printf("\t\t%s\n", everything[no][count[no]-1]);

}

void theTable(){

    int m,n;
    FILE * fp;
    int h;

    /* open the file for writing*/
    //fp = fopen ("/home/ubuntu/Desktop/correctTestCase/test3/output.txt","w");

    for(h=0;h<20;h++){
        printf("___");
    }

    for(m=1;m<36;m++){
;
        if(count[m]>0){

            namethe(m);

            for(n=0;n<count[m];n++)
            {
                int p = strlen(everything[m][n]);
```

Compiler Design Project Submission - 2

```
    if(n==0 && n==count[m]-1)
    {
        printf(" | %s\n",everything[m][n]);
    }
    else if(n==0)
    {printf(" | %s\n",everything[m][n]);}
    else if(n== count[m]-1)
    {printf("\t\t\t| %s\n",everything[m][n]);
    }
    else{
        printf("\t\t\t| %s\n",everything[m][n]);
    }

}

if(count[m]==1){
    printf("|");
}
for(h=0;h<20;h++){
    printf("___");
}

}

}

printf("\n\n\n");

}
```

Compiler Design Project Submission - 2

Explanation of Implementation:

File Structure:

help.h, *project.l*, *project.y*, *inp* should be brother files.

- *help.h* – an extension file for *project.l*
- *project.y* – the *yacc* file which contains the grammatical rules of the language(C here)
- *project.l* – the *lex* file which contains the tokenisation rules of the language(here C)
- *Inp* – the *c* file which is an input to the parsing.

Sequence of Commands:

- `yacc -v -d project.y`
- `lex project.l`
- `gcc -o a.out y.tab.c lex.yy.c -lfl -lm`
- `./a.out <inp`

Sequence of Output files:

- *y.tab.c* , *y.tab.h* and *y.output*
- *lex project.l*
- *a.out*
- *Final Compilation*

Brief Explanation of Lex File:

There are 35 types of tokens in total. These can be referred to the *help.h* file above. According to the rules in the *project.l* file, the input C file is tokenised into various token and is sent into *help.h* file, which has a method of storage.

'everything[36][50]' is the character array used for storing these tokens. The output is printed on the command prompt on execution, when the *.y* file calls the *Table()* from *help.h*

After the storage is done and the respective output is linked to *.y* – *yacc* file, then the tokens are noted and using the parsing rule as coded in the *project.y*, the parse tree is traversed.

This code is the hierarchy and precedence of the parse tree:

Compiler Design Project Submission - 2

```
%token int_const char_const float_const id string enumeration_const storage_const type_const
qual_const struct_const enum_const DEFINE
%token IF FOR DO WHILE BREAK SWITCH CONTINUE RETURN CASE DEFAULT GOTO
SIZEOF PUNC or_const and_const eq_const shift_const rel_const inc_const
%token point_const param_const ELSE HEADER
%left '+' '-'
%left '*' '/'
%nonassoc "then"
%nonassoc ELSE
%define parse.error verbose
%start program_unit
```

Using these rules above mentioned, the parse tree is traversed and these help them from solving ambiguities.

Although, not evrything is mentioned in the implemented program, this is the general definiton and method of implemetation.

Keyword	Description
%left	Identifies tokens that are left-associative with other tokens.
%nonassoc	Identifies tokens that are not associative with other tokens.
%right	Identifies tokens that are right-associative with other tokens.
%start	Identifies a nonterminal name for the start symbol.
%token	Identifies the token names that the yacc command accepts. Declares all token names in the declarations section.
%type	Identifies the type of nonterminals. Type-checking is performed when this construct is present.

Compiler Design Project Submission - 2

Keyword	Description
%union	Identifies the yacc value stack as the union of the various type of values desired. By default, the values returned are integers. The effect of this construct is to provide the declaration of YYSTYPE directly from the input.

The **%token**, **%left**, **%right**, and **%nonassoc** keywords optionally support the name of a C union member (as defined by **%union**) called a *<Tag>* (literal angle brackets surrounding a union member name). The **%type** keyword requires a *<Tag>*. The use of *<Tag>* specifies that the tokens named on the line are to be of the same C type as the union member referenced by *<Tag>*. For example, the following declaration declares the *Name* parameter to be a token:

```
%token [<Tag>] Name [Number] [Name [Number]]...
```

Now, after the whole process is complete, the output file is achieved as required.

TEST CASES

Test Case1:

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>
```

```
/* A binary tree node has data, pointer to left child
```

```
and a pointer to right child */
```

```
struct node
```

```
{
```

```
    int data;
```

```
    struct node* left;
```

```
    struct node* right;
```

```
};
```

Compiler Design Project Submission - 2

```
int isBSTUtil(struct node* node, int min, int max);

/* Returns true if the given tree is a binary search tree
(efficient version). */

int isBST(struct node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
values are >= min and <= max. */

int isBSTUtil(struct node* node, int min, int max)
{
    /* an empty tree is BST */
    if (node==NULL)
        return 1;

    /* false if this node violates the min/max constraint */
    if (node->data < min || node->data > max)
        return 0;

    /* otherwise check the subtrees recursively,
    tightening the min or max constraint */
    return
        isBSTUtil(node->left, min, node->data-1) && // Allow only distinct values
```

Compiler Design Project Submission - 2

```
isBSTUtil(node->right, node->data+1, max); // Allow only distinct values
}
```

```
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
```

```
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));

    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}
```

```
/* Driver program to test above functions*/
```

```
int main()
{
    float a=-2.0;

    struct node *root = newNode(4);

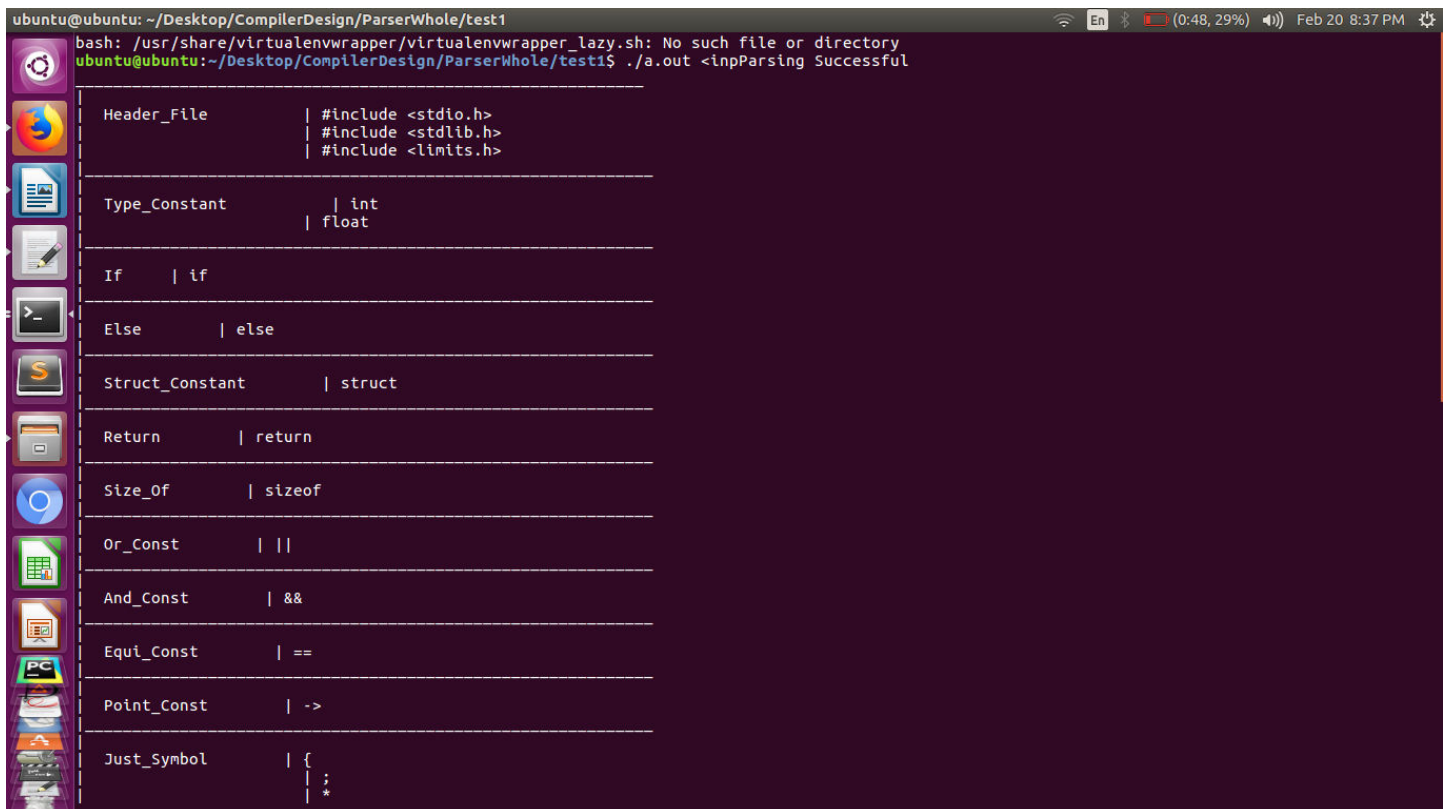
    root->left    = newNode(2);
    root->right   = newNode(5);
    root->left->left = newNode(1);
    root->left->right = newNode(3);
```


Compiler Design Project Submission - 2

```
if(isBST(root))  
  
    printf("Is BST");  
  
else  
  
    printf("Not a BST");  
  
  
getchar();  
  
return 0;  
  
}
```

Output:

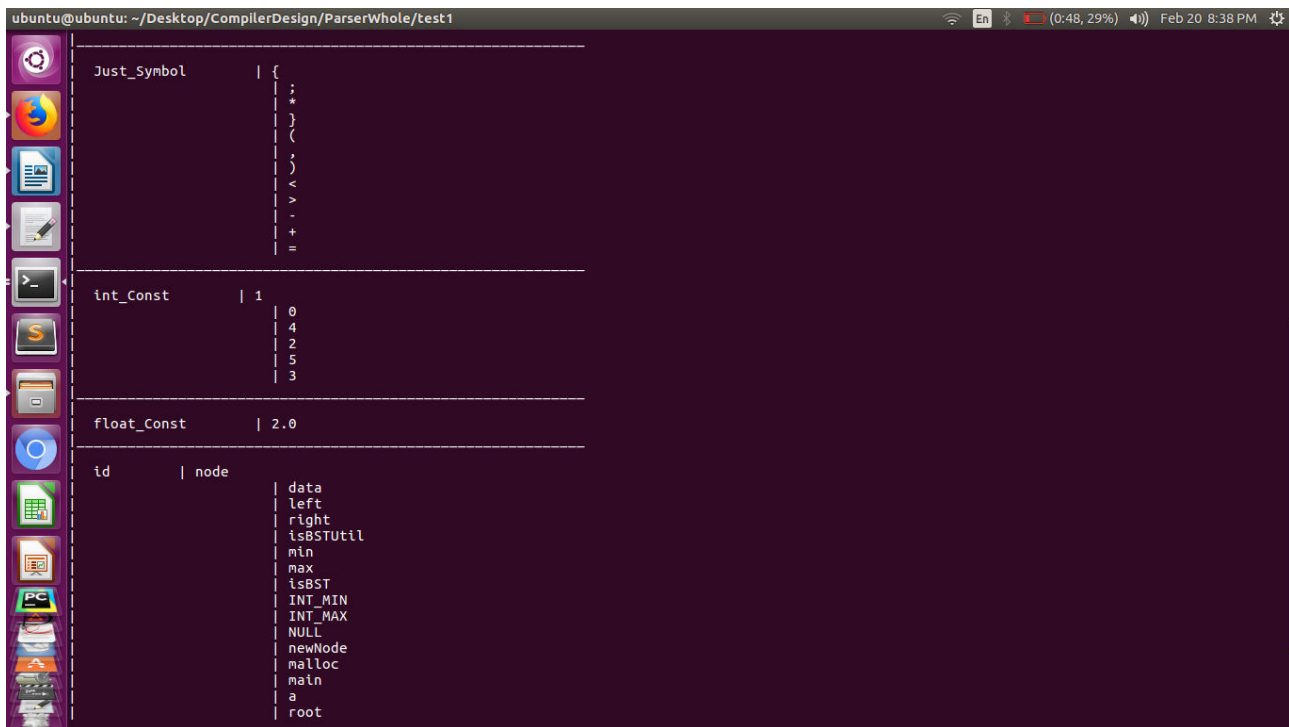
Parsing Successful



```
ubuntu@ubuntu: ~/Desktop/CompilerDesign/ParserWhole/test1
bash: /usr/share/virtualenvwrapper/virtualenvwrapper_lazy.sh: No such file or directory
ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test1$ ./a.out <inpParsing Successful

Header_File      | #include <stdio.h>
                  | #include <stdlib.h>
                  | #include <limits.h>
-----
Type_Constant    | int
                  | float
-----
If               | if
-----
Else             | else
-----
Struct_Constant  | struct
-----
Return           | return
-----
Size_Of          | sizeof
-----
Or_Const         | ||
-----
And_Const        | &&
-----
Equi_Const       | ==
-----
Point_Const      | ->
-----
Just_Symbol      | {
                  | ;
                  | *
```

Compiler Design Project Submission - 2



The screenshot shows a terminal window with a dark purple background. The title bar at the top reads 'ubuntu@ubuntu: ~/Desktop/CompilerDesign/ParserWhole/test1'. The terminal content is organized into sections separated by dashed lines. The first section, titled 'Just_Symbol', lists symbols: '{', ';', '*', '}', '(', ')', '<', '>', '-', '+', and '='. The second section, titled 'int_Const', lists the integer 1 followed by a list of integers: 0, 4, 2, 5, and 3. The third section, titled 'Float_Const', lists the float value 2.0. The fourth section, titled 'id', lists a 'node' structure with various fields: data, left, right, isBSTUtil, min, max, isBST, INT_MIN, INT_MAX, NULL, newNode, malloc, main, a, and root.

```
Just_Symbol | {
             | ;
             | *
             | }
             | (
             | )
             | <
             | >
             | -
             | +
             | =
-----
int_Const   | 1
             | 0
             | 4
             | 2
             | 5
             | 3
-----
Float_Const | 2.0
-----
id          | node
             | data
             | left
             | right
             | isBSTUtil
             | min
             | max
             | isBST
             | INT_MIN
             | INT_MAX
             | NULL
             | newNode
             | malloc
             | main
             | a
             | root
```

Test Case 2:

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

```
#include <stdlib.h>
```

```
#include <limits.h>
```

```
struct node
```

```
{
```

```
    int data;
```

```
    struct node* left;
```

```
    struct node* right;
```

```
};
```

```
int isBSTUtil(struct node* node, int min, int max)
```

```
/* Driver program to test above functions*/
```

Compiler Design Project Submission - 2

```
int main()

{float a=-2.0;

struct node *root = newNode(4);

root->left      = newNode(2);

root->right     = newNode(5);

root->left->left = newNode(1);

root->left->right = newNode(3);

if(isBST(root))

    printf("Is BST");

else

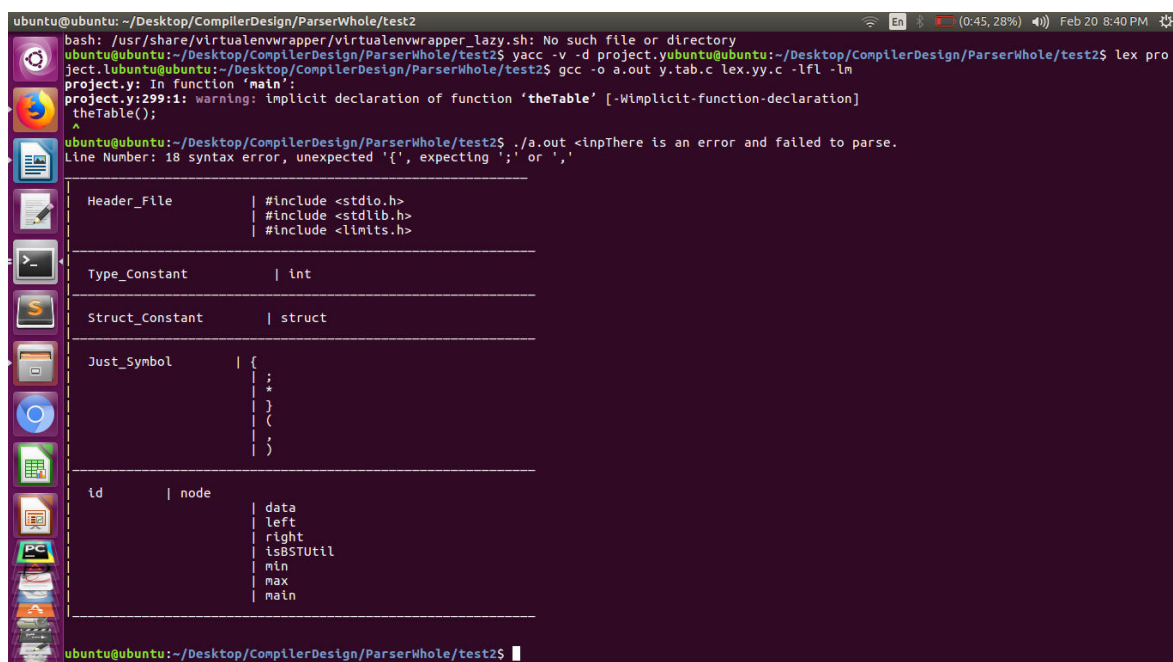
    printf("Not a BST")

getchar();

return 0;

}
```

Output: Unable to parse



The screenshot shows a terminal window with the following output:

```
ubuntu@ubuntu: ~/Desktop/CompilerDesign/ParserWhole/test2
bash: /usr/share/virtualenvwrapper/virtualenvwrapper_lazy.sh: No such file or directory
ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test2$ yacc -v -d project.y ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test2$ lex pro
ject.l ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test2$ gcc -o a.out y.tab.c lex.yy.c -lfl -lm
project.y: In function 'main':
project.y:299:1: warning: implicit declaration of function 'theTable' [-Wimplicit-function-declaration]
theTable();
^
ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test2$ ./a.out <inpThere is an error and failed to parse.
Line Number: 18 syntax error, unexpected '{', expecting ';' or ','
```

Header_File		#include <stdio.h>
		#include <stdlib.h>
		#include <limits.h>
Type_Constant		int
Struct_Constant		struct
Just_Symbol		{
		;
		*
		}
		(
		,
)
id		node
		data
		left
		right
		isBSTUtil
		min
		max
		main

ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test2\$

Compiler Design Project Submission - 2

Test Case 3:

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */

struct node
{
    int data;

    struct node* left;

    struct node* right;
};

int isBSTUtil(struct node* node, int min, int max);

/* Returns true if the given tree is a binary search tree
   (efficient version). */

int isBST(struct node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
```

Compiler Design Project Submission - 2

```
values are >= min and <= max. */
```

```
int isBSTUtil(struct node* node, int min, int max)
```

```
{
```

```
    /* an empty tree is BST */
```

```
    if (node==NULL)
```

```
        return 1;
```

```
    /* false if this node violates the min/max constraint */
```

```
    if (node->data < min || node->data > max)
```

```
        return 0;
```

```
    /* otherwise check the subtrees recursively,
```

```
    tightening the min or max constraint */
```

```
    return
```

```
        isBSTUtil(node->left, min, node->data-1) && // Allow only distinct values
```

```
        isBSTUtil(node->right, node->data+1, max); // Allow only distinct values
```

```
}
```

```
/* Helper function that allocates a new node with the
```

```
given data and NULL left and right pointers. */
```

```
struct node* newNode(int data)
```

```
{
```

```
    struct node* node = (struct node*)
```

```
        malloc(sizeof(struct node));
```

```
    node->data = data;
```

Compiler Design Project Submission - 2

```
node->left = NULL;

node->right = NULL;


return(node);
}


/* Driver program to test above functions*/

int main()
{
    float a=-2.0;

    struct node *root = newNode(4);

    root->left    = newNode(2);

    root->right    = newNode(5);

    root->left->left = newNode(1);

    root->left->right = newNode(3);


    if(isBST(root))

        printf("Is BST");

    else

        printf("Not a BST");


    getchar();

    return 0;
}
```

Compiler Design Project Submission - 2

Output: Unable to parse

```
ubuntu@ubuntu: ~/Desktop/CompilerDesign/ParserWhole/test3
tu:~/Desktop/CompilerDesign/ParserWhole/test3$ gcc -o a.out y.tab.c lex.yy.c -lfl -lm
project.y: In function 'main':
project.y:299:1: warning: implicit declaration of function 'theTable' [-Wimplicit-function-declaration]
theTable();
^
ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test3$ ./a.out <inpThere is a
n error and failed to parse.
Line Number: 3 syntax error, unexpected '<'

-----
Header_File      | #include <stdio.h>
                  | #include <stdlib.h>
-----
Just_Symbol      | <
-----
id               | include
-----

ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test3$
```

Test Case 4:

```
int main()
{
    float a=-2.0;

    struct node *root = newNode(4);

    root->left      = newNode(2);

    root->right     = newNode(5);

    root->left->left = newNode(1);

    root->left->right = newNode(3);
```

Compiler Design Project Submission - 2

```
if(isBST(root))  
  
    printf("Is BST");  
  
else  
  
    printf("Not a BST");  
  
  
getchar();  
  
return 0;  
  
}
```

Output : Parsing succesful for an incorrect program

```
ubuntu@ubuntu: ~/Desktop/CompilerDesign/ParserWhole/test4
bash: /usr/share/virtualenvwrapper/virtualenvwrapper_lazy.sh: No such file or directory
ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test4$ ./a.out <inp
Parsing Successful
```

Type_Constant		int
		float

If		if
----	--	----

Else		else
------	--	------

Struct_Constant		struct
-----------------	--	--------

Return		return
--------	--	--------

Point_Const		->
-------------	--	----

Just_Symbol		(
)
		{
		}
		=
		-
		+
		*
		}

int_Const		4
		2
		5
		1
		3
		0

float_Const		2.0
-------------	--	-----

float_Const		2.0
-------------	--	-----

id		main
		a
		node
		root
		newNode
		left
		right
		isBST
		printf
		getchar

string		"Is BST"
		"Not a BST"


```
ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test4$
```


Compiler Design Project Submission - 2

Test Case 5:

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

int main()

{

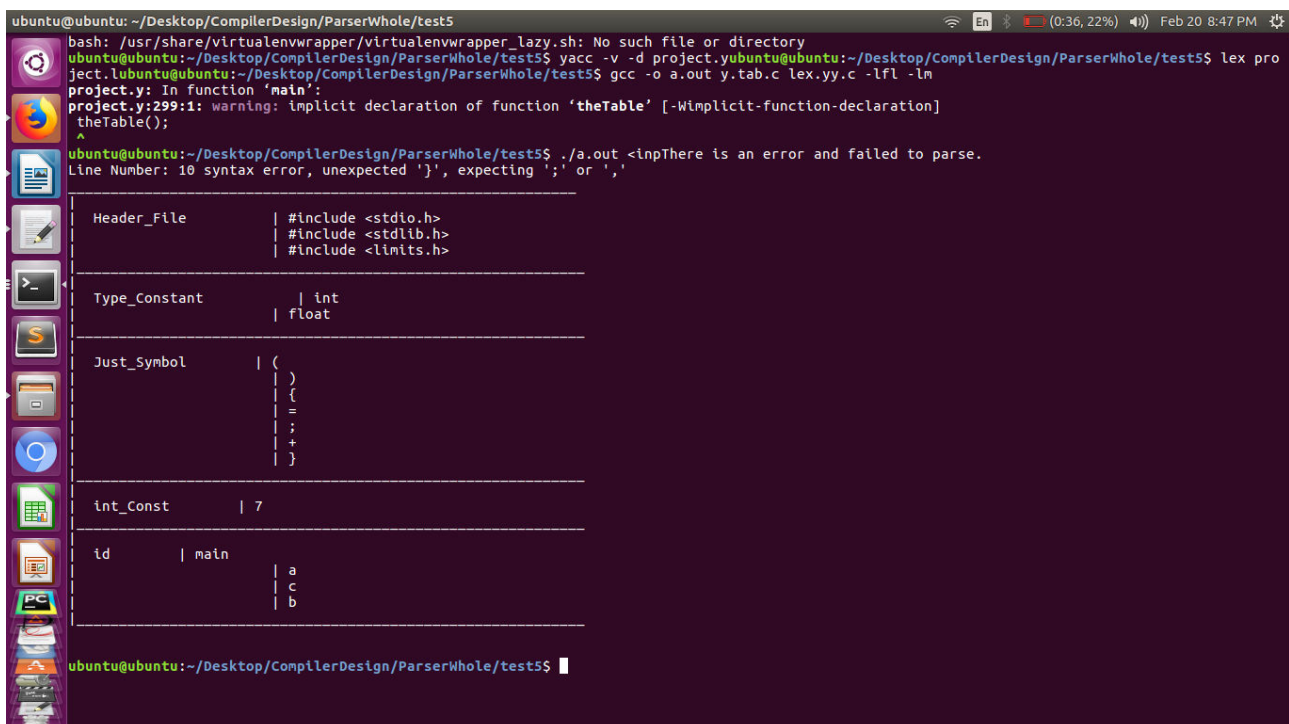
    float a=7;

    c= a+b

}
```

Output:

Unable to parse



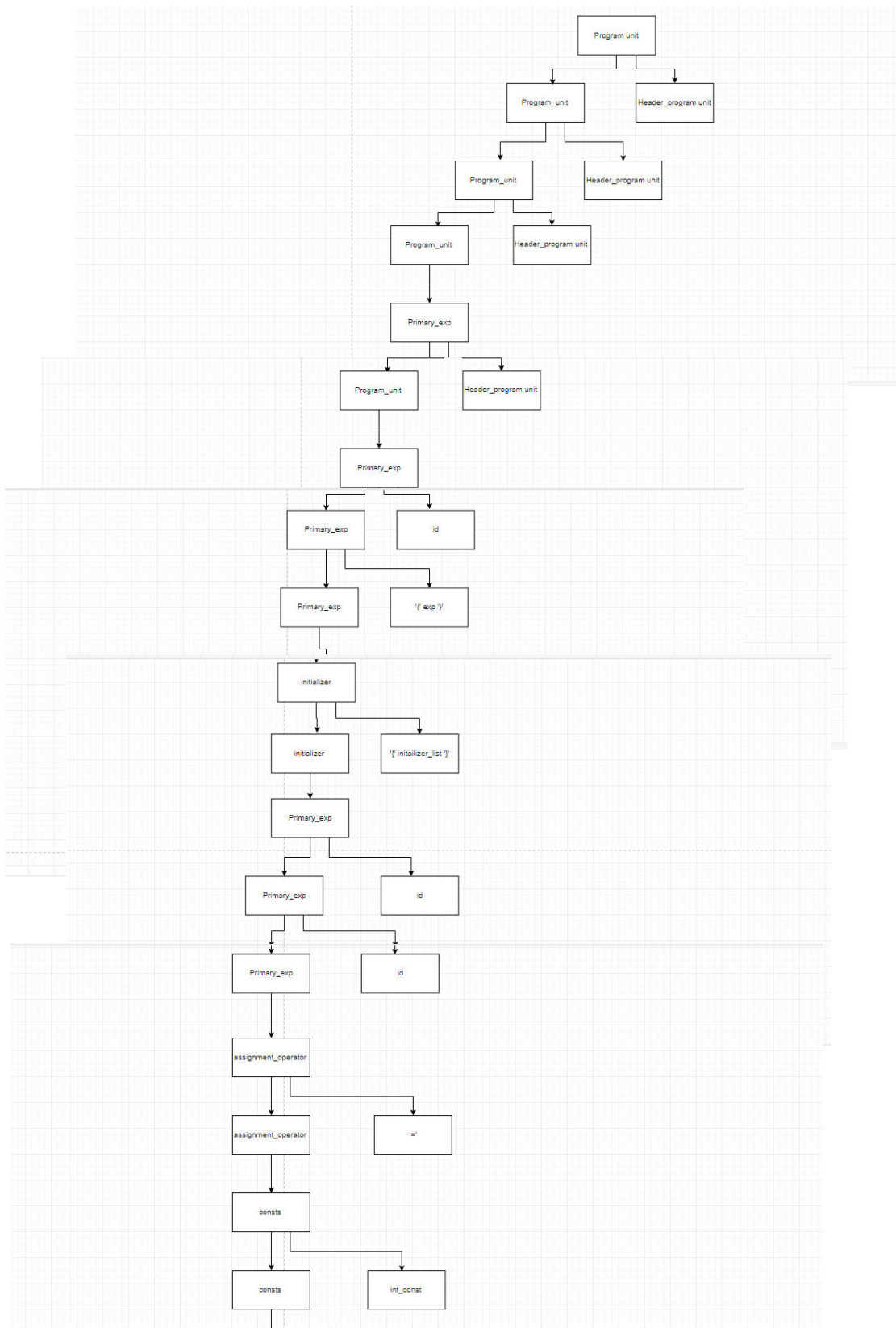
```
ubuntu@ubuntu: ~/Desktop/CompilerDesign/ParserWhole/test5
bash: /usr/share/virtualenvwrapper/virtualenvwrapper_lazy.sh: No such file or directory
ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test5$ yacc -v -d project.y
ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test5$ lex pro
project.l: In function 'main':
project.y:299:1: warning: implicit declaration of function 'theTable' [-Wimplicit-function-declaration]
theTable();
^
ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test5$ ./a.out <inp
There is an error and failed to parse.
Line Number: 10 syntax error, unexpected '}', expecting ';' or ','
```

Header_File	#include <stdio.h> #include <stdlib.h> #include <limits.h>
Type_Constant	int float
Just_Symbol	({ = ; + }
int_Const	7
id	main a c b

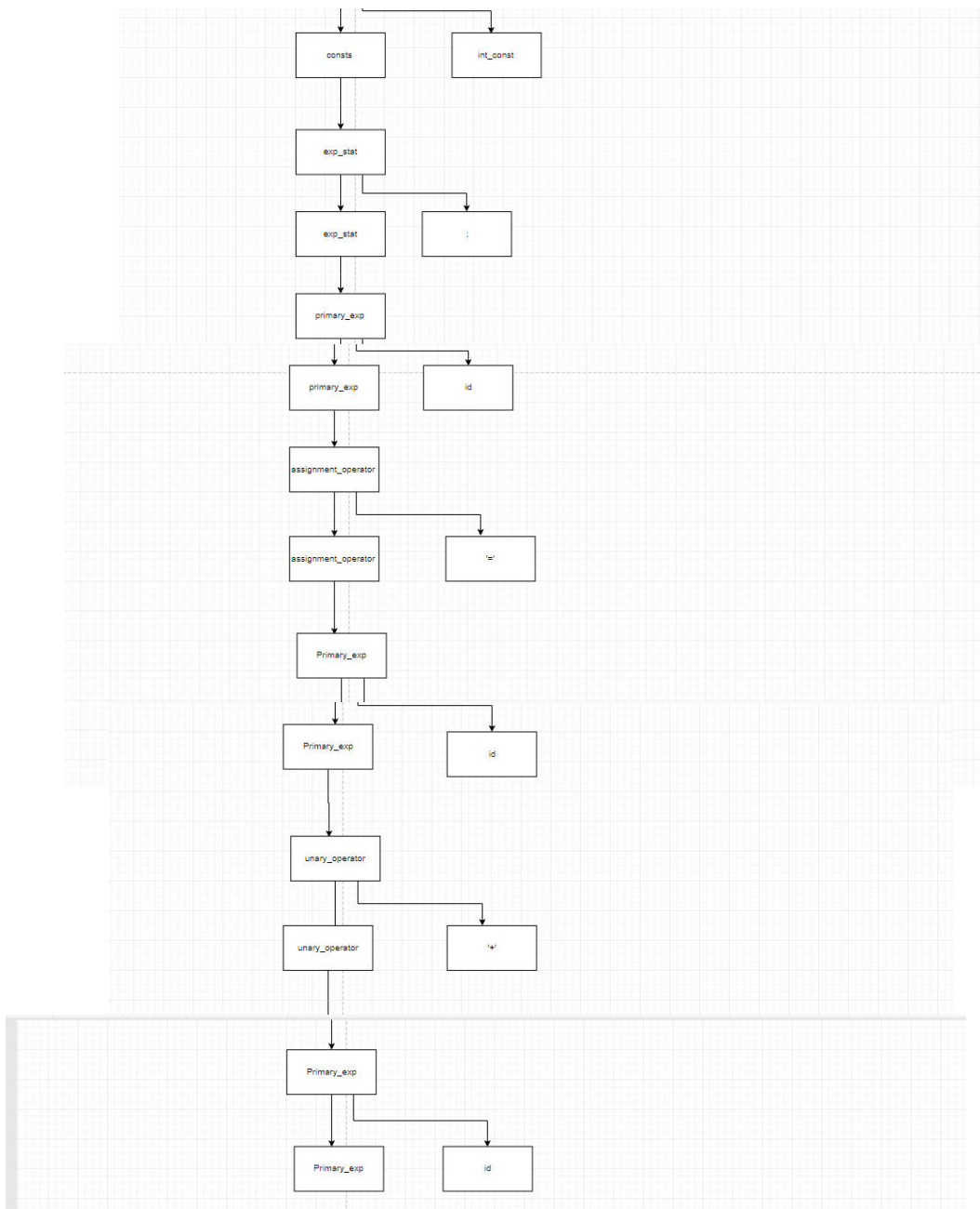
```
ubuntu@ubuntu:~/Desktop/CompilerDesign/ParserWhole/test5$
```

Compiler Design Project Submission - 2

Parse tree of the test case 5:



Compiler Design Project Submission - 2



Conclusion:

In order to understand the information supplied to a text parsing application, there are two phases. The first is simply to identify what has been typed or provided to an application. You must be able to identify the key words, phrases, or character sequences from the input source so that you can determine what to do with them. The second process is to understand the structure of that information -- the grammar -- so that the input can be both validated and operated on. An excellent example of grammar is the use of parentheses in most programming languages. Hence, using lex and yacc files, it checks errors in the input code and produces the output and the parse tree. The tokenisation is done using lex as seen in screen shots and parse tree using yacc files as shown above.