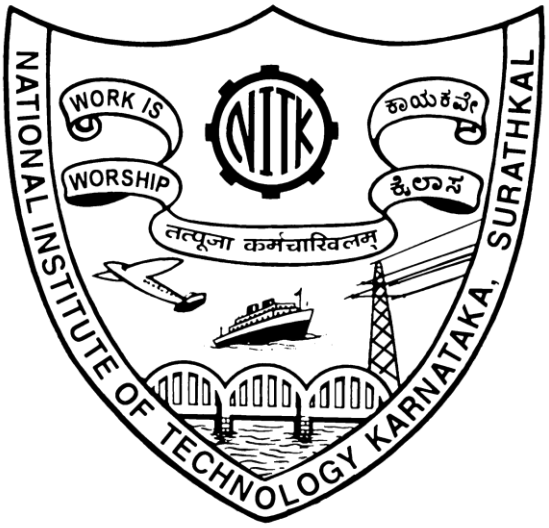
Syntax Analyser for the C Language



National Institute of Technology Karnataka Surathkal

Date: 08/02/2017

Submitted To: Uma Priya

Group Members:

Sheetal Shalini (14CO142)

Navya R S (14CO126)

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, making sure that statements that refer to other statements are referred to correctly in the final 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 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. 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.

Contents:

- Introduction ---------- 1
    - Syntax Analyser ---------- 1
    - Flex Script ---------- 1
    - Yacc Script ---------- 3
- Design of Programs ---------- 6
    - Code ---------- 6
    - Explanation ---------- 14
- Test Cases ---------- 16
    - Without Errors ---------- 16
    - With Errors ---------- 21
- Implementation ---------- 29
- Shift-Reduce Conflicts ---------- 29
- Results ---------- 30
- Future work ---------- 30
- References ---------- 30

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

The script written by us is a program that generates syntax analyzers (parsers). Lex reads an input stream specifying the syntax analyzer and outputs source code implementing the parser in the C programming language.

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 may contain the following items.

- 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.

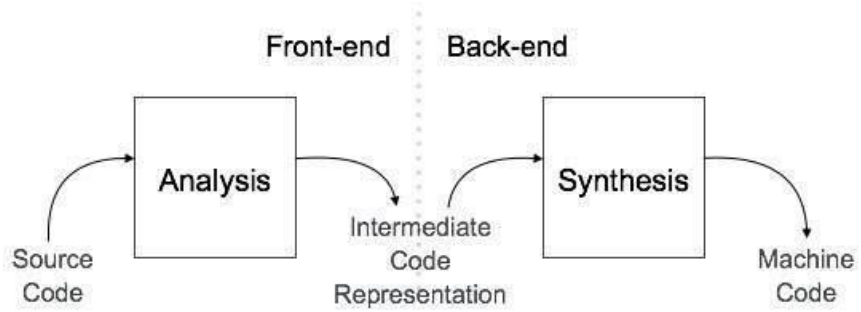
C Program

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 in 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 an 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.

Design of Programs

Flow :



Parser checks the C code for syntactical errors and specifies them if present. Otherwise, it displays ‘parsing complete’.

Codes:

Lex Code : (parser.l file)

```
alpha           [A-Za-z_]
fl          (f|F|l|L)
ul          (u|U|l|L)*
digit           [0-9]
space           [ ]
hex         [a-fA-F0-9]
exp         [Ee][+-]?{digit}+

%{
int yylineno;
#include <stdio.h>
%}

%%
\n   { yylineno++; }
"/*"                    { multicomment(); }
"//"                    { singlecomment(); }
"#include<"({alpha})*".h>" {}
"#define"({space})""({alpha})""({alpha}|{digit})*""({space})""({digit})+"" { return DEF;}
"#define"({space})""({alpha}({alpha}|{digit})*)""({space})""(({digit}+)\.({digit}+))"" { return DEF;}
"#define"({space})""({alpha}({alpha}|{digit})*)""({space})""({alpha}({alpha}|{digit})*)"" { return DEF;}
{digit}+                    {  return CONSTANT; }
({digit}+)\.({digit}+)          {  return CONSTANT; }
0[xX]{hex}+{ul}?            {  return CONSTANT; }
0{digit}+{ul}?              {  return CONSTANT; }
{digit}+{ul}?               {  return CONSTANT; }
{alpha}?'(\\.|[^\\'])+'         {  return CONSTANT; }
{digit}+{exp}{fl}?          {  return CONSTANT; }
{digit}*"."{digit}+({exp})?{fl}?    {  return CONSTANT; }
{digit}+"."{digit}*({exp})?{fl}?    {  return CONSTANT; }
{alpha}?\"(\\.|[^\\"])*\"   {  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_ASSIGN; }
"/="            {  return DIV_ASSIGN; }
"%="            {  return MOD_ASSIGN; }
"+="            {  return ADD_ASSIGN; }
"-="            {  return SUB_ASSIGN; }
"<<="           {  return LEFT_ASSIGN; }
">>="           {  return RIGHT_ASSIGN; }
"&="            {  return AND_ASSIGN; }
"^="            {  return XOR_ASSIGN; }
"|="            {  return OR_ASSIGN; }
"typedef"       {  return TYPEDEF; }
"extern"        {  return EXTERN; }
```

```
"static"        {  return STATIC; }
"auto"          {  return AUTO; }
"register"      {  return REGISTER; }
"char"          {  return CHAR; }
"short"         {  return SHORT; }
"int"           {  return INT; }
"long"          {  return LONG; }
"signed"        {  return SIGNED; }
"unsigned"      {  return UNSIGNED; }
"float"         {  return FLOAT; }
"double"        {  return DOUBLE; }
"const"         {  return CONST; }
"volatile"      {  return VOLATILE; }
"void"          {  return VOID; }
"struct"        {  return STRUCT; }
"union"         {  return UNION; }
"enum"          {  return ENUM; }
"..."           {  return ELLIPSIS; }
"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('?'); }
{alpha}({alpha}|{digit})*           {  return IDENTIFIER; }
[ \t\v\n\f]     {  }
.           { /* ignore bad 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)

```
%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_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN 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 ELLIPSIS
%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN
%nonassoc ELSE
%start translation_unit
%glr-parser
%%
primary_expression
   : IDENTIFIER
   | CONSTANT
   | STRING_LITERAL
   | '(' expression ')'
   | Define primary_expression
   ;
```

```
Define
    : DEF ;

postfix_expression
    : primary_expression
    | postfix_expression '[' expression ']'
    | postfix_expression '(' ')'
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression '.' IDENTIFIER
    | postfix_expression PTR_OP IDENTIFIER
    | postfix_expression INC_OP
    | postfix_expression DEC_OP
    ;

argument_expression_list
    : assignment_expression
    | argument_expression_list ',' assignment_expression
    ;

unary_expression
    : postfix_expression
    | INC_OP unary_expression
    | DEC_OP unary_expression
    | unary_operator cast_expression
    | SIZEOF unary_expression
    | SIZEOF '(' type_name ')'
    ;

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

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 assignment_operator assignment_expression
    ;

assignment_operator
    : '='
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
```

```
    | SUB_ASSIGN
    | LEFT_ASSIGN
    | RIGHT_ASSIGN
    | AND_ASSIGN
    | XOR_ASSIGN
    | OR_ASSIGN
    ;

expression
    : assignment_expression
    | expression ',' assignment_expression
    ;

constant_expression
    : conditional_expression
    ;

declaration
    : declaration_specifiers ';'
    | declaration_specifiers init_declarator_list ';'
    ;

declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier declaration_specifiers
    | type_specifier
    | type_specifier 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_specifier
    : TYPEDEF
    | EXTERN
    | STATIC
    | AUTO
    | REGISTER
    ;

type_specifier
    : 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
    : type_specifier specifier_qualifier_list
    | type_specifier
    | 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 ',' ELLIPSIS
    ;

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
    | iteration_statement
    | 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
    ;

iteration_statement
    : 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
    | stmnt 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
    ;

stmnt: error ';'

%%
#include"lex.yy.c"
#include<ctype.h>
#include <stdio.h>
int main(int argc, char *argv[])
```

```
{
   yyin = fopen(argv[1], "r");
   if(!yyparse())
      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);
}
```

Explanation :

Files :

1. **Parser.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.
2. **Parser.y** : Yacc file is where the productions for all the loops, selection and conditional statements and expressions are mentioned.
3. Test.c : 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
- Single-line comments
  Statements processed : *//.....*
  Token generated : Single Line Comment
- Multi-line comments
  Statements processed : */*...../*.../*...*/*
  Token generated : Multi Line Comment
- Nested comments
  Statements processed : */*...../*..../*..../*....*/*
  Token generated : Error with line number

- Operators

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.

- 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

The yacc Script specifies productions for the following:

- Primary expressions
- Argument expressions list
- Unary expressions
- Postfix expressions

- 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)

  Statements processed : *(..), {..}, [..] (without errors)*

  (..)..), {..}..}, [..]..], (..., {..., [... (with errors)

  Tokens generated : Parenthesis (without error) / Error with line number (witherror)

Test Cases:

Without Errors:

1. Code : (Nested if-else)

```
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
   int a=4;
   if(a<10)
   {
      a=a+1;
      printf("\n%d\n",a);
   }
```

```
    else if(a<5)
    {
        a=a+2;
    }
    else
    {
        a=a-2;
    }
    return 0;
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Parsing complete
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

2. Code (Nested while)

```
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
    int a=4;
    while(a<10)
    {
        printf("%d",a);
        j=1;
        while(j<=4)
            j++;
    }
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Parsing complete
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

3. Code (Nested do-while)

```
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
    int a=4;
    do
    {
        printf("%d",a);
        j=1;
        do
        {
            j++;
        }while(j<=4);
```

```
   }while(a<10);
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Parsing complete
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

4. Code (Nested for loop)

```
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
   int a=4;
   for(i=0;i<10;i++)
   {
      printf("%d",i);
      for(j=i;j<=8;j++)
         a++;
   }
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Parsing complete
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

5. Code (Nested switch statement)

```
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
   int a=4,i=1;
   switch(a)
   {
      case 1 : switch(i)
           {
             case 2 : printf("%d",i);
                   break;
             case 4 : printf("4");
           }
             break;
      case 6 : printf("6");
      default : printf("def");
   }
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Parsing complete
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

6. Code (Structures)

```
#include<stdio.h>
#define x 3
struct s
{
   int a,b;
}st;
int main(int argc,int *argv[])
{
   int c=4;
   printf("%d",st.a);
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Parsing complete
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

7. Code (Union)

```
#include<stdio.h>
#define x 3
union s
{
   int a,b;
}st;
int main(int argc,int *argv[])
{
   int c=4;
   printf("%d",st.a);
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Parsing complete
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

8. Code (Single & Multi-line comments)

```
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
   //int c=4;
```

```
    int b=5;
    /* hello
       bye*/
    printf("%d",b);
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Parsing complete
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

With Errors :

1. Code (Missing semicolon, multiple errors)

```
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
    int a=4
    while(a<10)
    {
        a=a+1;
        printf("%d\n",a)
    }
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Line 6 : syntax error

Line 9 : syntax error

Parsing failed
```

2. Code (Nested if-else dangling else error)

```
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
    int a=4;
    if(a<10)
        printf("10");
    else
    {
        if(a<12)
            printf("11");
        else
            printf("All");
        else
            printf("error");
```

```
    }
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Line 14 : syntax error

Parsing failed
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

3. Code (Nested while loop error)

```
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
    int a=4;
    whle(a<10)
    {
        printf("%d",a);
        j=1;
        while(j<=4)
            j++;
    }
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Line 7 : syntax error

Parsing failed
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

4. Code (Nested do-while loop error)

```
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
    int a=4;
    do
    {
        printf("%d",a);
        j=1;
        do;
        {
            j++;
        }while(j<=4);
    }while(a<10);
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Line 11 : syntax error

Parsing failed
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

5. Code (Nested for loop error)

```
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
   int a=4;
   for(i=0;i<10)
   {
      printf("%d",i);
      for(j=i;j<=8;j++)
         a++;
   }
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Line 6 : syntax error

Parsing failed
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

6. Code (Nested switch statement error)

```
#include<stdio.h>
#define x 3
int main(int argc,int *argv[])
{
   int a=4,i=1;
   switch(a)
   {
      case 1 : switch(i)
           {
             case 2 : printf("%d",i);
                   break;
             case 4 : printf("4");
           }
             break;
      case 6  printf("6");
      default : printf("def");
   }
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Line 15 : syntax error

Parsing failed
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

7. Code (Structures error)

```
#include<stdio.h>
#define x 3
struct s
{
   int a,b;
}st
int main(int argc,int *argv[])
{
   int c=4;
   printf("%d",st.a);
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Line 8 : syntax error

Parsing failed
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

8. Code (Union error)

```
#include<stdio.h>
#define x 3
union s
{
   int a,b;
}st;
int main(int argc,int *argv[])
{
   int c=4;
   printf("%d",st->.a);
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$ ./a.out test.c

Line 10 : syntax error

Parsing failed
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/Syntax$
```

Implementation

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

- **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 (parser.l) and yacc (parser.y) codes are compiled and executed by the following terminal commands to parse the given input file (test.c)

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

After parsing, if there are errors then the line numbers of those errors are displayed along with a ‘parsing failed’on the terminal. Otherwise, a ‘parsing complete’ message is displayed on the console.

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 disambiguating rules by default:

1. 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 favor 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 parsers 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 you want to declare a precedence for the token, you need to use %left, %right or %nonassoc all of which both declare the token AND set its precedence.

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
        ;
```

Possibilities of Shift/reduce conflicts and handling them:

1. 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
        | 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 was first encountered in the specifications of Algol 60 and is called the “dangling else” ambiguity.

2. 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. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. 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 keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators that may not associate with themselves.

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

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

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

2. 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.

3. 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:
Shift-reduce conflicts are harmless as they are anyways handled by the yacc.

Shift-reduce conflicts are normally not errors. The conflict is resolved by always doing the shift which is usually what you want. Most or all real-world grammars have shift-reduce conflicts. And if you wanted the reduction you can arrange for that with precedence declarations.

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.

Shift-reduce conflict occurs when the parser is faced with a choice of a shift action and a reduce action. Yacc’s default action in the case of a shift-reduce conflict is to choose the shift action.

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.

Future work

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

References

1. Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

2. Lex and Yacc By John R. Levine, Tony Mason, Doug Brown

3. https://www.programiz.com/c-programming/precedence-associativity-operators