

**Parser for C- language**

**Course Project - 2 (July 2020 - Dec 2020)**

**CS305 Compiler Design Lab**

**National Institute of Technology, Karnataka**



**SUBMITTED TO:**

Prof. P. Santhi Thilagam

CSE Dept, NITK  
  
**TEAM**

Kesana Jahnavi (181CO127)

Shreeya Sanjay Sand (181CO150)

Shumbul Arifa (181CO152)

Keerti Chaudhary (181CO226)

**Abstract**

This report contains the details of the tasks finished as a part of Phase Two of Compiler Design Lab. We have developed a Parser for C language which makes use of the C lexer to parse the given C input file. The parser code has the functionality of taking input through a file or through standard input. This makes it more user-friendly and efficient at the same time. Parsing, syntax analysis, or syntactic analysis is the process of analyzing 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).

Within computational linguistics the term is used to refer to the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a parse tree showing their syntactic relation to each other, which may also contain semantic and other information (p-values). Some parsing algorithms may generate a parse forest or list of parse trees for a syntactically ambiguous input.

In some machine translation and natural language processing systems, written texts in human languages are parsed by computer programs. Human sentences are not easily parsed by programs, as there is substantial ambiguity in the structure of human language, whose usage is to convey meaning (or semantics) amongst a potentially unlimited range of possibilities but only some of which are germane to the particular case. So an utterance "Man bites dog" versus "Dog bites man" is definite on one detail but in another language might appear as "Man dog bites" with a reliance on the larger context to distinguish between those two possibilities, if indeed that difference was of concern. It is difficult to prepare formal rules to describe informal behaviour even though it is clear that some rules are being followed.

The parser checks whether the generated tokens form a meaningful expression. This makes use of a context-free grammar that defines algorithmic procedures for components. These work to form an expression and define the particular order in which tokens must be placed. It generates a list of identifiers and functions with their types and also specifies syntax errors if any.

INDEX

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No** | **CONTENT** | | **PAGE NUMBER** |
| 1 | Abstract | | 1 |
| 2 | Overview | Features | 3 |
| Results | 3 |
| Tools Used | 3 |
| 3 | Introduction | Parser Analysis | 3 |
| Yacc Script | 4 |
| C Program | 5 |
| 4 | Design of Programs | Code | 5 |
| Explanation | 29 |
| 5 | Test Cases | Without errors | 31 |
| With errors | 33 |
| 6 | Implementation | Handling Conflicts | 35 |
| Dangling Else Problem | 36 |
| 7 | Parse Tree | | 36 |
| 8 | Conclusion | | 37 |
| 9 | Future Work | | 37 |
| 10 | References | | 37 |

**List of tables:**

**Table 1 : 5 test cases without errors**

**Table 2 : 5 test cases with errors**

Overview

**FEATURES**

The project objective is to construct a compiler that studies the C programming language. It will have the following features:

* Check the syntax of variable declaration of int, float and char type and also short, long, signed, unsigned subtypes.
* Declaration of arrays with specified datatype (eg: int arr[10])
* Declaration of looping constructs such as while, nested while.
* Proper usage of looping constructs such as while, nested while.
* Declaration and definition of function with one or no argument.
* Check parenthesis balancing
* Precedence and associativity of the operators
* Hashing techniques used to maintain symbol and constant tables.
* Appropriate error messages with line number are displayed.

**RESULTS**

* Errors in the source program along with appropriate error messages
* Symbol table and constant table will be designed using hashing organization techniques.

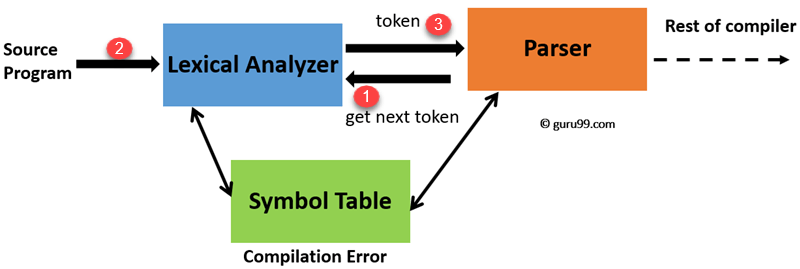
**TOOLS USED**

* Flex
* Yacc

Introduction

**PARSER/SYNTACTIC ANALYSIS**

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. The parser analyses the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

[](https://www.guru99.com/images/1/020819_1105_LexicalAnal1.png)

There are three general types of parsers for grammars: universal, top-down, and bottom-up. As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time. The most efficient top-down and bottom-up methods work only for sub-classes of grammars, but several of these classes, particularly, LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages. Parsers implemented by hand often use LL grammars. Parsers for the larger class of LR grammars are usually constructed using automated tools

**YACC SCRIPT**

The parser written is known as the Yacc program. The structure of the Yacc file is similar to that of the lexer, consisting of three sections:

{declarations}

%%

{rules}

%%

{routines}

The **declarations section** of a yacc file may consist of the following:

* %token - identifies the token names that the yacc file accepts
* %start - identifies a nonterminal name for the start symbol
* %right - identifies tokens that are right-associative with other tokens
* %left - identifies tokens that are left-associative with other tokens
* %nonassoc - identifies tokens that are not associative with other tokens

The **rules section** consists of the context free grammar used to generate the parse tree. A general rule has the following structure:

nonterminal

: sentential form

| sentential form

.................

| sentential form

;

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

The **routines section** may include the C program that specifies the input file, action routines and other user defined functions.

**C PROGRAM**

The workflow is explained as follows:

* Compile the script using Yacc tool.

$ yacc -d parser.y

* Compile the flex script using Flex tool.

$ flex lexer.l

* After compiling the lex file, lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.
* Compilation is done with the options –ll, –ly and -w.

$ gcc -o compiler lex.yy.c y.tab.h y.tab.c -ll -ly

* The executable file is generated, which on running parses the C file given as a command line input.

$ ./a.out test1.c

Code

Lexer Code

|  |
| --- |
| %{ *#include <stdio.h>* *#include <string.h>* *#include <stdlib.h>* *#include "y.tab.h"* struct tokenList {  char \*token,type[20],line[100];  struct tokenList \*next; }; typedef struct tokenList tokenList;  extern FILE \*yyin; extern int lineCount; extern char \*tablePtr; extern int nestedCommentCount; extern int commentFlag;  char typeBuffer=' ';  tokenList \*symbolPtr = NULL; tokenList \*constantPtr = NULL; tokenList \*parsedPtr=NULL;  char \*sourceCode=NULL; int errorFlag=0; void makeList(char \*,char,int); %}  %token  AUTO BREAK  CASE CHAR  CONST  CONTINUE  DEFAULT  DO DOUBLE  ELSE ENUM  %token EXTERN FLOAT  FOR GOTO  IF INT LONG REGISTER  RETURN SHORT SIGNED   %token SIZEOF STATIC STRUCT SWITCH TYPEDEF UNION UNSIGNED VOID VOLATILE WHILE     %token IDENTIFIER  %token CONSTANT STRING\_LITERAL  %token ELLIPSIS  %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  %nonassoc LOWER\_THAN\_ELSE %nonassoc ELSE  %start translation\_unit  %%  primary\_expression  : IDENTIFIER  { makeList(tablePtr, 'v', lineCount); }  | CONSTANT    { makeList(tablePtr, 'c', lineCount);}  | STRING\_LITERAL  { makeList(tablePtr, 's', lineCount);}  | '(' expression ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  postfix\_expression  : primary\_expression  | postfix\_expression '[' expression ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | postfix\_expression '(' ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | postfix\_expression '(' argument\_expression\_list ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | postfix\_expression '.' IDENTIFIER { makeList(tablePtr, 'v', lineCount);}  | postfix\_expression PTR\_OP IDENTIFIER { makeList(tablePtr, 'v', lineCount);}  | postfix\_expression INC\_OP  { makeList(tablePtr, 'o', lineCount);}  | postfix\_expression DEC\_OP  { makeList(tablePtr, 'o', lineCount);}  ;  argument\_expression\_list  : assignment\_expression  | argument\_expression\_list ',' assignment\_expression { makeList(",",'p', lineCount); }  ;  unary\_expression  : postfix\_expression  | INC\_OP unary\_expression { makeList("++",'o', lineCount); }  | DEC\_OP unary\_expression { makeList("--",'o', lineCount); }  | unary\_operator cast\_expression  | SIZEOF unary\_expression { makeList("sizeof",'o', lineCount); }  | SIZEOF '(' type\_name ')' { makeList("sizeof",'o', lineCount); }   { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  unary\_operator  : '&' { makeList("&",'o', lineCount); }  | '\*' { makeList("\*",'o', lineCount); }  | '+' { makeList("+",'o', lineCount); }  | '-' { makeList("-",'o', lineCount); }  | '~' { makeList("~",'o', lineCount); }  | '!' { makeList("!",'o', lineCount); }  ;  cast\_expression  : unary\_expression  | '(' type\_name ')' cast\_expression { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  multiplicative\_expression  : cast\_expression  | multiplicative\_expression '\*' cast\_expression { makeList("\*",'o', lineCount); }  | multiplicative\_expression '/' cast\_expression { makeList("/",'o', lineCount); }  | multiplicative\_expression '%' cast\_expression { makeList("%",'o', lineCount); }  ;  additive\_expression  : multiplicative\_expression  | additive\_expression '+' multiplicative\_expression { makeList("+",'o', lineCount); }  | additive\_expression '-' multiplicative\_expression { makeList("-",'o', lineCount); }  ;  shift\_expression  : additive\_expression  | shift\_expression LEFT\_OP additive\_expression { makeList("<<",'o', lineCount); }  | shift\_expression RIGHT\_OP additive\_expression { makeList(">>",'o', lineCount); }  ;  relational\_expression  : shift\_expression  | relational\_expression '<' shift\_expression  | relational\_expression '>' shift\_expression  | relational\_expression LE\_OP shift\_expression { makeList("<=",'o', lineCount); }  | relational\_expression GE\_OP shift\_expression { makeList(">=",'o', lineCount); }  ;  equality\_expression  : relational\_expression  | equality\_expression EQ\_OP relational\_expression { makeList("==",'o', lineCount); }  | equality\_expression NE\_OP relational\_expression { makeList("!=",'o', lineCount); }  ;  and\_expression  : equality\_expression  | and\_expression '&' equality\_expression { makeList("&", 'o', lineCount);}  ;  exclusive\_or\_expression  : and\_expression  | exclusive\_or\_expression '^' and\_expression { makeList("^", 'o', lineCount); }  ;  inclusive\_or\_expression  : exclusive\_or\_expression  | inclusive\_or\_expression '|' exclusive\_or\_expression { makeList("|", 'o', lineCount); }  ;  logical\_and\_expression  : inclusive\_or\_expression  | logical\_and\_expression AND\_OP inclusive\_or\_expression { makeList("&&", 'o', lineCount); }  ;  logical\_or\_expression  : logical\_and\_expression  | logical\_or\_expression OR\_OP logical\_and\_expression { makeList("||", 'o', lineCount); }  ;  conditional\_expression  : logical\_or\_expression  | logical\_or\_expression '?' expression ':' conditional\_expression { makeList("?:",'o', lineCount); }  ;  assignment\_expression  : conditional\_expression  | unary\_expression assignment\_operator assignment\_expression  ;  assignment\_operator  : '=' { makeList("=",'o', lineCount); }  | MUL\_ASSIGN { makeList("\*=",'o', lineCount); }  | DIV\_ASSIGN { makeList("/=",'o', lineCount); }  | MOD\_ASSIGN { makeList("%=",'o', lineCount); }  | ADD\_ASSIGN { makeList("+=",'o', lineCount); }  | SUB\_ASSIGN { makeList("-=",'o', lineCount); }  | LEFT\_ASSIGN { makeList("<<=",'o', lineCount); }  | RIGHT\_ASSIGN { makeList(">==",'o', lineCount); }  | AND\_ASSIGN { makeList("&=",'o', lineCount); }  | XOR\_ASSIGN { makeList("^=",'o', lineCount); }  | OR\_ASSIGN { makeList("|=",'o', lineCount); }  ;  expression  : assignment\_expression  | expression ',' assignment\_expression { makeList(",", 'p', lineCount); }  ;  constant\_expression  : conditional\_expression  ;  declaration  : declaration\_specifiers ';'   { makeList(";", 'p', lineCount);typeBuffer=' '; }  | declaration\_specifiers init\_declarator\_list ';' { makeList(";", 'p', lineCount); typeBuffer=' ';}  ;  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 { makeList(",", 'p', lineCount); }  ;  init\_declarator  : declarator  | declarator '=' initializer { makeList("=", 'o', lineCount); }  ;  storage\_class\_specifier  : TYPEDEF { makeList("typedef", 'k', lineCount);}  | EXTERN { makeList("extern", 'k', lineCount);}  | STATIC { makeList("static", 'k', lineCount);}  | AUTO { makeList("auto", 'k', lineCount);}  | REGISTER { makeList("register", 'k', lineCount);}  ;  type\_specifier  : VOID { makeList("void", 'k', lineCount);typeBuffer='v';}  | CHAR { makeList("char", 'k', lineCount); typeBuffer='c';}  | SHORT { makeList("short", 'k', lineCount);}  | INT { makeList("int", 'k', lineCount); typeBuffer='i';}  | LONG { makeList("lon``g", 'k', lineCount);}  | FLOAT { makeList("float", 'k', lineCount); typeBuffer='f';}  | DOUBLE { makeList("double", 'k', lineCount);}  | SIGNED { makeList("signed", 'k', lineCount);}  | UNSIGNED { makeList("unsigned", 'k', lineCount);}  | 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 { makeList("struct", 'k', lineCount);}  | UNION { makeList("union", 'k', lineCount);}  ;  struct\_declaration\_list  : struct\_declaration  | struct\_declaration\_list struct\_declaration  ;  struct\_declaration  : specifier\_qualifier\_list struct\_declarator\_list ';' { makeList(";", 'p', lineCount); }  ;  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 { makeList(",", 'p', lineCount); }  ;  struct\_declarator  : declarator  | ':' constant\_expression { makeList(":", 'p', lineCount); }  | declarator ':' constant\_expression { makeList(":", 'p', lineCount); }  ;  enum\_specifier  : ENUM '{' enumerator\_list '}' { makeList("enum", 'k', lineCount);}  | ENUM IDENTIFIER '{' enumerator\_list '}' { makeList("enum", 'k', lineCount); makeList(tablePtr, 'v', lineCount); }  | ENUM IDENTIFIER { makeList("enum", 'k', lineCount); makeList(tablePtr, 'v', lineCount); }  ;  enumerator\_list  : enumerator  | enumerator\_list ',' enumerator { makeList(",", 'p', lineCount); }  ;  enumerator  : IDENTIFIER { makeList(tablePtr, 'v', lineCount); }  | IDENTIFIER '=' constant\_expression { makeList("=", 'o', lineCount); makeList("tablePtr", 'v', lineCount); }  ;  type\_qualifier  : CONST { makeList("const", 'k', lineCount); }  | VOLATILE { makeList("volatile", 'k', lineCount); }  ;  declarator  : pointer direct\_declarator  | direct\_declarator  ;  direct\_declarator  : IDENTIFIER { makeList(tablePtr, 'v', lineCount); }  | '(' declarator ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | direct\_declarator '[' constant\_expression ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | direct\_declarator '[' ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | direct\_declarator '(' parameter\_type\_list ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | direct\_declarator '(' identifier\_list ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | direct\_declarator '(' ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  pointer  : '\*' { makeList("\*", 'o', lineCount); }  | '\*' type\_qualifier\_list { makeList("\*", 'o', lineCount); }  | '\*' pointer { makeList("\*", 'o', lineCount); }  | '\*' type\_qualifier\_list pointer { makeList("\*", 'o', lineCount); }  ;  type\_qualifier\_list  : type\_qualifier  | type\_qualifier\_list type\_qualifier  ;   parameter\_type\_list  : parameter\_list  | parameter\_list ',' ELLIPSIS { makeList(",", 'p', lineCount); makeList("::", 'o', lineCount); }  ;  parameter\_list  : parameter\_declaration  | parameter\_list ',' parameter\_declaration { makeList(",", 'p', lineCount); }  ;  parameter\_declaration  : declaration\_specifiers declarator  | declaration\_specifiers abstract\_declarator  | declaration\_specifiers  ;  identifier\_list  : IDENTIFIER {makeList(tablePtr, 'v', lineCount);}  | identifier\_list ',' IDENTIFIER { makeList(tablePtr, 'v', lineCount); makeList(",", 'p', lineCount); }  ;  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 ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | '[' ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | '[' constant\_expression ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | direct\_abstract\_declarator '[' ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | direct\_abstract\_declarator '[' constant\_expression ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | '(' ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | '(' parameter\_type\_list ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | direct\_abstract\_declarator '(' ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | direct\_abstract\_declarator '(' parameter\_type\_list ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  initializer  : assignment\_expression  | '{' initializer\_list '}'  | '{' initializer\_list ',' '}'  ;  initializer\_list  : initializer  | initializer\_list ',' initializer { makeList(",", 'p', lineCount); }  ;  statement  : labeled\_statement  | compound\_statement  | expression\_statement  | selection\_statement  | iteration\_statement  | jump\_statement  ;  labeled\_statement  : IDENTIFIER ':' statement  { makeList(tablePtr, 'v', lineCount);  }  | CASE constant\_expression ':'  statement { makeList(":", 'p', lineCount); makeList("case", 'k', lineCount);}  | DEFAULT ':' statement { makeList(":", 'p', lineCount); makeList("default", 'k', lineCount); }  ;  compound\_statement  : '{' '}'  | '{' statement\_list '}'  | '{' declaration\_list '}'  | '{' declaration\_list statement\_list '}'  ;  declaration\_list  : declaration  | declaration\_list declaration  ;  statement\_list  : statement  | statement\_list statement  ;  expression\_statement  : ';' { makeList(";", 'p', lineCount); }  | expression ';' { makeList(";", 'p', lineCount); }  ;  selection\_statement  : IF '(' expression ')' statement %prec LOWER\_THAN\_ELSE   { makeList("if", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount);}   | IF '(' expression ')' statement ELSE statement    { makeList("if", 'k', lineCount);  makeList("else", 'k', lineCount); makeList("(", 'p', lineCount);   makeList(")", 'p', lineCount);    }  | SWITCH '(' expression ')' statement   { makeList("switch", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  iteration\_statement  : WHILE '(' expression ')' statement    { makeList("while", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | DO statement WHILE '(' expression ')' ';'   { makeList("do", 'k', lineCount); makeList("while", 'k', lineCount); makeList("(", 'p', lineCount);           makeList(")", 'p', lineCount); makeList(";", 'p', lineCount);   }  | FOR '(' expression\_statement expression\_statement ')' statement    { makeList("for", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | FOR '(' expression\_statement expression\_statement expression ')' statement   { makeList("for", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  jump\_statement  : GOTO IDENTIFIER ';' { makeList("goto", 'k', lineCount); makeList(";", 'p', lineCount); makeList(tablePtr, 'v', lineCount);}  | CONTINUE ';' { makeList("continue", 'k', lineCount); makeList(";", 'p', lineCount); }  | BREAK ';'  { makeList("break", 'k', lineCount); makeList(";", 'p', lineCount);}  | RETURN ';'  { makeList("return", 'k', lineCount); makeList(";", 'p', lineCount);}  | RETURN expression ';' { makeList("return", 'k', lineCount); makeList(";", 'p', lineCount);}  ;  translation\_unit  : external\_declaration  | translation\_unit external\_declaration  ;  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  ;  %% void yyerror() {  errorFlag=1;  fflush(stdout);  printf("\n%s : %d :Syntax error \n",sourceCode,lineCount); } void main(int argc,char \*\*argv){  if(argc<=1){    printf("Invalid ,Expected Format : ./a.out <\"sourceCode\"> \n");  return 0;  }    yyin=fopen(argv[1],"r");  sourceCode=(char \*)malloc(strlen(argv[1])\*sizeof(char));  sourceCode=argv[1];  yyparse();    if(nestedCommentCount!=0){  errorFlag=1;     printf("%s : %d : Comment Does Not End\n",sourceCode,lineCount);       }  if(commentFlag==1){  errorFlag=1;  printf("%s : %d : Nested Comment\n",sourceCode,lineCount);     }   if(!errorFlag){    printf("\n\n\t\t%s Parsing Completed\n\n",sourceCode);   FILE \*writeFile=fopen("Output.txt","w"); fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");  fprintf(writeFile,"\n\t\t\t\t\t\t\t\tPARSED TABLE\n");  fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");    fprintf(writeFile,"\t\tIdentifier \t\t\t\tToken\t\t\t\t\t\t Line Number\n");   fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");  for(tokenList \*ptr=parsedPtr;ptr!=NULL;ptr=ptr->next){   fprintf(writeFile,"\n%20s%40s%60s",ptr->token,ptr->type,ptr->line);  }   fprintf(writeFile,"\n \n \n");  fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");  fprintf(writeFile,"\n\t\t\t\t\t\t\t\tSYMBOL TABLE\n");  fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");    fprintf(writeFile,"\tIdentifier \t\t\t\t\tType\t\t\t\t\t\t Line Number\n");   fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");  for(tokenList \*ptr=symbolPtr;ptr!=NULL;ptr=ptr->next){   fprintf(writeFile,"\n%20s%40s%50s",ptr->token,ptr->type,ptr->line);  }   fprintf(writeFile,"\n \n \n");  fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");  fprintf(writeFile,"\n\t\t\t\t\t\t\t\tCONSTANT TABLE\n");  fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");    fprintf(writeFile,"\t\tConstant Value \t\t\t\t\t Line Number\n");   fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");  for(tokenList \*ptr=constantPtr;ptr!=NULL;ptr=ptr->next){   fprintf(writeFile,"\n%30s%50s",ptr->token,ptr->line);  }  fclose(writeFile);  } printf("\n\n");  }  void makeList(char \*tokenName,char tokenType, int tokenLine) {  char line[39],lineBuffer[19];     snprintf(lineBuffer, 19, "%d", tokenLine);  strcpy(line," ");  strcat(line,lineBuffer);  char type[20];  switch(tokenType)  {  case 'c':  strcpy(type,"Constant");  break;  case 'v':  strcpy(type,"Identifier");  break;  case 'p':  strcpy(type,"Punctuator");  break;  case 'o':  strcpy(type,"Operator");  break;  case 'k':  strcpy(type,"Keyword");  break;  case 's':  strcpy(type,"String Literal");  break;  case 'd':  strcpy(type,"Preprocessor Statement");  break;  }  for(tokenList \*p=parsedPtr;p!=NULL;p=p->next)   if(strcmp(p->token,tokenName)==0){        strcat(p->line,line);        goto xx;      }  tokenList \*temp=(tokenList \*)malloc(sizeof(tokenList));  temp->token=(char \*)malloc(strlen(tokenName)+1);  strcpy(temp->token,tokenName);  strcpy(temp->type,type);     strcpy(temp->line,line);     temp->next=NULL;          tokenList \*p=parsedPtr;     if(p==NULL){          parsedPtr=temp;     }     else{     while(p->next!=NULL){     p=p->next;     }     p->next=temp;     }          xx:  if(tokenType == 'c')  {          for(tokenList \*p=constantPtr;p!=NULL;p=p->next)   if(strcmp(p->token,tokenName)==0){        strcat(p->line,line);        return;      }  tokenList \*temp=(tokenList \*)malloc(sizeof(tokenList));  temp->token=(char \*)malloc(strlen(tokenName)+1);  strcpy(temp->token,tokenName);  strcpy(temp->type,type);     strcpy(temp->line,line);     temp->next=NULL;          tokenList \*p=constantPtr;     if(p==NULL){     constantPtr=temp;     }     else{     while(p->next!=NULL){     p=p->next;     }     p->next=temp;     }         }  if(tokenType=='v')  {     for(tokenList \*p=symbolPtr;p!=NULL;p=p->next)   if(strcmp(p->token,tokenName)==0){        strcat(p->line,line);        return;      }  tokenList \*temp=(tokenList \*)malloc(sizeof(tokenList));  temp->token=(char \*)malloc(strlen(tokenName)+1);  strcpy(temp->token,tokenName);  switch(typeBuffer){  case 'i': strcpy(temp->type,"INT"); break;  case 'f': strcpy(temp->type,"FLOAT");break;  case 'v' :strcpy(temp->type,"VOID");break;  case 'c': strcpy(temp->type,"CHAR");break;    }       strcpy(temp->line,line);     temp->next=NULL;     tokenList \*p=symbolPtr;     if(p==NULL){          symbolPtr=temp;     }     else{     while(p->next!=NULL){     p=p->next;     }     p->next=temp;     }  } }     %{ *#include <stdio.h>* *#include "y.tab.h"* int lineCount=1; int nestedCommentCount=0; int commentFlag=0; char \*tablePtr; void addToken(char\*); %}  digit [0-9] letter [a-zA-Z\_] hex [a-fA-F0-9] E [Ee][+-]?{digit}+ FS (f|F|l|letter) IS (u|U|l|letter)\*  singlelineComment (\/\/.\*) multilineCommentStart (\/\\*) multilineCommentEnd (\\*\/) %x DETECT\_COMMENT  %%   {singlelineComment}      { }  {multilineCommentStart}            { BEGIN(DETECT\_COMMENT);    nestedCommentCount++;   }   <DETECT\_COMMENT>{multilineCommentStart} { nestedCommentCount++;    if(nestedCommentCount>1)    commentFlag = 1;  }   <DETECT\_COMMENT>{multilineCommentEnd} { BEGIN(INITIAL);   if(nestedCommentCount>0)                                 nestedCommentCount--;                             if(nestedCommentCount==0)                             BEGIN(INITIAL);                     }                      <DETECT\_COMMENT>\n {lineCount++;} <DETECT\_COMMENT>. {}     "auto" {  return(AUTO); } "break" {  return(BREAK); } "case" {  return(CASE); } "char" {  return(CHAR); } "const" {  return(CONST); } "continue" {  return(CONTINUE); } "default" {  return(DEFAULT); } "do" {  return(DO); } "double" {  return(DOUBLE); } "else" {  return(ELSE); } "enum" {  return(ENUM); } "extern" {  return(EXTERN); } "float" {  return(FLOAT); } "for" {  return(FOR); } "goto" {  return(GOTO); } "if" {  return(IF); } "int" {  return(INT); } "long" {  return(LONG); } "register" {  return(REGISTER); } "return" {  return(RETURN); } "short" {  return(SHORT); } "signed" {  return(SIGNED); } "sizeof" {  return(SIZEOF); } "static" {  return(STATIC); } "struct" {  return(STRUCT); } "switch" {  return(SWITCH); } "typedef" {  return(TYPEDEF); } "union" {  return(UNION); } "unsigned" {  return(UNSIGNED); } "void" {  return(VOID); } "volatile" {  return(VOLATILE); } "while" {  return(WHILE); }  {letter}({letter}|{digit})\* {  addToken(yytext); return(IDENTIFIER); } {letter}?\"(\\.|[^\\"])\*\" {  addToken(yytext); return(STRING\_LITERAL); }  0[xX]{hex}+{IS}? {  addToken(yytext); return(CONSTANT); } 0{digit}+{IS}? {  addToken(yytext); return(CONSTANT); } {hex}+{IS}? {  addToken(yytext); return(CONSTANT); } {letter}?'(\\.|[^\\'])+' {  addToken(yytext); return(CONSTANT); } {digit}+{E}{FS}? {  addToken(yytext); return(CONSTANT); } {digit}\*"."{digit}+({E})?{FS}? {  addToken(yytext); return(CONSTANT); } {digit}+"."{digit}\*({E})?{FS}? {  addToken(yytext); return(CONSTANT); }  "..." {  return(ELLIPSIS); } ">>=" {  return(RIGHT\_ASSIGN); } "<<=" {  return(LEFT\_ASSIGN); } "+=" {  return(ADD\_ASSIGN); } "-=" {  return(SUB\_ASSIGN); } "\*=" {  return(MUL\_ASSIGN); } "/=" {  return(DIV\_ASSIGN); } "%=" {  return(MOD\_ASSIGN); } "&=" {  return(AND\_ASSIGN); } "^=" {  return(XOR\_ASSIGN); } "|=" {  return(OR\_ASSIGN); } ">>" {  return(RIGHT\_OP); } "<<" {  return(LEFT\_OP); } "++" {  return(INC\_OP); } "--" {  return(DEC\_OP); } "->" {  return(PTR\_OP); } "&&" {  return(AND\_OP); } "||" {  return(OR\_OP); } "<=" {  return(LE\_OP); } ">=" {  return(GE\_OP); } "==" {  return(EQ\_OP); } "!=" {  return(NE\_OP); } ";" {  return(';'); } ("{"|"<%") {  makeList("{",'p',lineCount); return('{'); } ("}"|"%>") {  makeList("}",'p',lineCount); return('}'); } "," {  return(','); } ":" {  return(':'); } "=" {  return('='); } "(" {  return('('); } ")" {  return(')'); } ("["|"<:") {  return('['); } ("]"|":>") {  return(']'); } "." {  return('.'); } "&" {  return('&'); } "!" {  return('!'); } "~" {  return('~'); } "-" {  return('-'); } "+" {  return('+'); } "\*" {  return('\*'); } "/" {  return('/'); } "%" {  return('%'); } "<" {  return('<'); } ">" {  return('>'); } "^" {  return('^'); } "|" {  return('|'); } "?" {  return('?'); }   "*#include"(.)\*"\n" { lineCount++; }* "#define"(.)\*"\n" { lineCount++; }  [ ] {} [\t\v\f] {} [\n] { lineCount++; } . { }  %% yywrap() {  return(1); }  void addToken(char \*yytext) {   int len = strlen(yytext);   tablePtr = (char\*)malloc((len+1)\*sizeof(char));   strcpy(tablePtr, yytext); } |

Parser Code

|  |
| --- |
| %{ *#include <stdio.h>* *#include <string.h>* *#include <stdlib.h>* *#include "y.tab.h"* struct tokenList {  char \*token,type[20],line[100];  struct tokenList \*next; }; typedef struct tokenList tokenList;  extern FILE \*yyin; extern int lineCount; extern char \*tablePtr; extern int nestedCommentCount; extern int commentFlag;  char typeBuffer=' ';  tokenList \*symbolPtr = NULL; tokenList \*constantPtr = NULL; tokenList \*parsedPtr=NULL;  char \*sourceCode=NULL; int errorFlag=0; void makeList(char \*,char,int); %}  %token  AUTO BREAK  CASE CHAR  CONST  CONTINUE  DEFAULT  DO DOUBLE  ELSE ENUM  %token EXTERN FLOAT  FOR GOTO  IF INT LONG REGISTER  RETURN SHORT SIGNED   %token SIZEOF STATIC STRUCT SWITCH TYPEDEF UNION UNSIGNED VOID VOLATILE WHILE     %token IDENTIFIER  %token CONSTANT STRING\_LITERAL  %token ELLIPSIS  %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  %nonassoc LOWER\_THAN\_ELSE %nonassoc ELSE  %start translation\_unit  %%  primary\_expression  : IDENTIFIER  { makeList(tablePtr, 'v', lineCount); }  | CONSTANT    { makeList(tablePtr, 'c', lineCount);}  | STRING\_LITERAL  { makeList(tablePtr, 's', lineCount);}  | '(' expression ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  postfix\_expression  : primary\_expression  | postfix\_expression '[' expression ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | postfix\_expression '(' ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | postfix\_expression '(' argument\_expression\_list ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | postfix\_expression '.' IDENTIFIER { makeList(tablePtr, 'v', lineCount);}  | postfix\_expression PTR\_OP IDENTIFIER { makeList(tablePtr, 'v', lineCount);}  | postfix\_expression INC\_OP  { makeList(tablePtr, 'o', lineCount);}  | postfix\_expression DEC\_OP  { makeList(tablePtr, 'o', lineCount);}  ;  argument\_expression\_list  : assignment\_expression  | argument\_expression\_list ',' assignment\_expression { makeList(",",'p', lineCount); }  ;  unary\_expression  : postfix\_expression  | INC\_OP unary\_expression { makeList("++",'o', lineCount); }  | DEC\_OP unary\_expression { makeList("--",'o', lineCount); }  | unary\_operator cast\_expression  | SIZEOF unary\_expression { makeList("sizeof",'o', lineCount); }  | SIZEOF '(' type\_name ')' { makeList("sizeof",'o', lineCount); }   { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  unary\_operator  : '&' { makeList("&",'o', lineCount); }  | '\*' { makeList("\*",'o', lineCount); }  | '+' { makeList("+",'o', lineCount); }  | '-' { makeList("-",'o', lineCount); }  | '~' { makeList("~",'o', lineCount); }  | '!' { makeList("!",'o', lineCount); }  ;  cast\_expression  : unary\_expression  | '(' type\_name ')' cast\_expression { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  multiplicative\_expression  : cast\_expression  | multiplicative\_expression '\*' cast\_expression { makeList("\*",'o', lineCount); }  | multiplicative\_expression '/' cast\_expression { makeList("/",'o', lineCount); }  | multiplicative\_expression '%' cast\_expression { makeList("%",'o', lineCount); }  ;  additive\_expression  : multiplicative\_expression  | additive\_expression '+' multiplicative\_expression { makeList("+",'o', lineCount); }  | additive\_expression '-' multiplicative\_expression { makeList("-",'o', lineCount); }  ;  shift\_expression  : additive\_expression  | shift\_expression LEFT\_OP additive\_expression { makeList("<<",'o', lineCount); }  | shift\_expression RIGHT\_OP additive\_expression { makeList(">>",'o', lineCount); }  ;  relational\_expression  : shift\_expression  | relational\_expression '<' shift\_expression  | relational\_expression '>' shift\_expression  | relational\_expression LE\_OP shift\_expression { makeList("<=",'o', lineCount); }  | relational\_expression GE\_OP shift\_expression { makeList(">=",'o', lineCount); }  ;  equality\_expression  : relational\_expression  | equality\_expression EQ\_OP relational\_expression { makeList("==",'o', lineCount); }  | equality\_expression NE\_OP relational\_expression { makeList("!=",'o', lineCount); }  ;  and\_expression  : equality\_expression  | and\_expression '&' equality\_expression { makeList("&", 'o', lineCount);}  ;  exclusive\_or\_expression  : and\_expression  | exclusive\_or\_expression '^' and\_expression { makeList("^", 'o', lineCount); }  ;  inclusive\_or\_expression  : exclusive\_or\_expression  | inclusive\_or\_expression '|' exclusive\_or\_expression { makeList("|", 'o', lineCount); }  ;  logical\_and\_expression  : inclusive\_or\_expression  | logical\_and\_expression AND\_OP inclusive\_or\_expression { makeList("&&", 'o', lineCount); }  ;  logical\_or\_expression  : logical\_and\_expression  | logical\_or\_expression OR\_OP logical\_and\_expression { makeList("||", 'o', lineCount); }  ;  conditional\_expression  : logical\_or\_expression  | logical\_or\_expression '?' expression ':' conditional\_expression { makeList("?:",'o', lineCount); }  ;  assignment\_expression  : conditional\_expression  | unary\_expression assignment\_operator assignment\_expression  ;  assignment\_operator  : '=' { makeList("=",'o', lineCount); }  | MUL\_ASSIGN { makeList("\*=",'o', lineCount); }  | DIV\_ASSIGN { makeList("/=",'o', lineCount); }  | MOD\_ASSIGN { makeList("%=",'o', lineCount); }  | ADD\_ASSIGN { makeList("+=",'o', lineCount); }  | SUB\_ASSIGN { makeList("-=",'o', lineCount); }  | LEFT\_ASSIGN { makeList("<<=",'o', lineCount); }  | RIGHT\_ASSIGN { makeList(">==",'o', lineCount); }  | AND\_ASSIGN { makeList("&=",'o', lineCount); }  | XOR\_ASSIGN { makeList("^=",'o', lineCount); }  | OR\_ASSIGN { makeList("|=",'o', lineCount); }  ;  expression  : assignment\_expression  | expression ',' assignment\_expression { makeList(",", 'p', lineCount); }  ;  constant\_expression  : conditional\_expression  ;  declaration  : declaration\_specifiers ';'   { makeList(";", 'p', lineCount);typeBuffer=' '; }  | declaration\_specifiers init\_declarator\_list ';' { makeList(";", 'p', lineCount); typeBuffer=' ';}  ;  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 { makeList(",", 'p', lineCount); }  ;  init\_declarator  : declarator  | declarator '=' initializer { makeList("=", 'o', lineCount); }  ;  storage\_class\_specifier  : TYPEDEF { makeList("typedef", 'k', lineCount);}  | EXTERN { makeList("extern", 'k', lineCount);}  | STATIC { makeList("static", 'k', lineCount);}  | AUTO { makeList("auto", 'k', lineCount);}  | REGISTER { makeList("register", 'k', lineCount);}  ;  type\_specifier  : VOID { makeList("void", 'k', lineCount);typeBuffer='v';}  | CHAR { makeList("char", 'k', lineCount); typeBuffer='c';}  | SHORT { makeList("short", 'k', lineCount);}  | INT { makeList("int", 'k', lineCount); typeBuffer='i';}  | LONG { makeList("lon``g", 'k', lineCount);}  | FLOAT { makeList("float", 'k', lineCount); typeBuffer='f';}  | DOUBLE { makeList("double", 'k', lineCount);}  | SIGNED { makeList("signed", 'k', lineCount);}  | UNSIGNED { makeList("unsigned", 'k', lineCount);}  | 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 { makeList("struct", 'k', lineCount);}  | UNION { makeList("union", 'k', lineCount);}  ;  struct\_declaration\_list  : struct\_declaration  | struct\_declaration\_list struct\_declaration  ;  struct\_declaration  : specifier\_qualifier\_list struct\_declarator\_list ';' { makeList(";", 'p', lineCount); }  ;  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 { makeList(",", 'p', lineCount); }  ;  struct\_declarator  : declarator  | ':' constant\_expression { makeList(":", 'p', lineCount); }  | declarator ':' constant\_expression { makeList(":", 'p', lineCount); }  ;  enum\_specifier  : ENUM '{' enumerator\_list '}' { makeList("enum", 'k', lineCount);}  | ENUM IDENTIFIER '{' enumerator\_list '}' { makeList("enum", 'k', lineCount); makeList(tablePtr, 'v', lineCount); }  | ENUM IDENTIFIER { makeList("enum", 'k', lineCount); makeList(tablePtr, 'v', lineCount); }  ;  enumerator\_list  : enumerator  | enumerator\_list ',' enumerator { makeList(",", 'p', lineCount); }  ;  enumerator  : IDENTIFIER { makeList(tablePtr, 'v', lineCount); }  | IDENTIFIER '=' constant\_expression { makeList("=", 'o', lineCount); makeList("tablePtr", 'v', lineCount); }  ;  type\_qualifier  : CONST { makeList("const", 'k', lineCount); }  | VOLATILE { makeList("volatile", 'k', lineCount); }  ;  declarator  : pointer direct\_declarator  | direct\_declarator  ;  direct\_declarator  : IDENTIFIER { makeList(tablePtr, 'v', lineCount); }  | '(' declarator ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | direct\_declarator '[' constant\_expression ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | direct\_declarator '[' ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | direct\_declarator '(' parameter\_type\_list ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | direct\_declarator '(' identifier\_list ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | direct\_declarator '(' ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  pointer  : '\*' { makeList("\*", 'o', lineCount); }  | '\*' type\_qualifier\_list { makeList("\*", 'o', lineCount); }  | '\*' pointer { makeList("\*", 'o', lineCount); }  | '\*' type\_qualifier\_list pointer { makeList("\*", 'o', lineCount); }  ;  type\_qualifier\_list  : type\_qualifier  | type\_qualifier\_list type\_qualifier  ;   parameter\_type\_list  : parameter\_list  | parameter\_list ',' ELLIPSIS { makeList(",", 'p', lineCount); makeList("::", 'o', lineCount); }  ;  parameter\_list  : parameter\_declaration  | parameter\_list ',' parameter\_declaration { makeList(",", 'p', lineCount); }  ;  parameter\_declaration  : declaration\_specifiers declarator  | declaration\_specifiers abstract\_declarator  | declaration\_specifiers  ;  identifier\_list  : IDENTIFIER {makeList(tablePtr, 'v', lineCount);}  | identifier\_list ',' IDENTIFIER { makeList(tablePtr, 'v', lineCount); makeList(",", 'p', lineCount); }  ;  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 ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | '[' ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | '[' constant\_expression ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | direct\_abstract\_declarator '[' ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | direct\_abstract\_declarator '[' constant\_expression ']' { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }  | '(' ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | '(' parameter\_type\_list ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | direct\_abstract\_declarator '(' ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | direct\_abstract\_declarator '(' parameter\_type\_list ')' { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  initializer  : assignment\_expression  | '{' initializer\_list '}'  | '{' initializer\_list ',' '}'  ;  initializer\_list  : initializer  | initializer\_list ',' initializer { makeList(",", 'p', lineCount); }  ;  statement  : labeled\_statement  | compound\_statement  | expression\_statement  | selection\_statement  | iteration\_statement  | jump\_statement  ;  labeled\_statement  : IDENTIFIER ':' statement  { makeList(tablePtr, 'v', lineCount);  }  | CASE constant\_expression ':'  statement { makeList(":", 'p', lineCount); makeList("case", 'k', lineCount);}  | DEFAULT ':' statement { makeList(":", 'p', lineCount); makeList("default", 'k', lineCount); }  ;  compound\_statement  : '{' '}'  | '{' statement\_list '}'  | '{' declaration\_list '}'  | '{' declaration\_list statement\_list '}'  ;  declaration\_list  : declaration  | declaration\_list declaration  ;  statement\_list  : statement  | statement\_list statement  ;  expression\_statement  : ';' { makeList(";", 'p', lineCount); }  | expression ';' { makeList(";", 'p', lineCount); }  ;  selection\_statement  : IF '(' expression ')' statement %prec LOWER\_THAN\_ELSE   { makeList("if", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount);}   | IF '(' expression ')' statement ELSE statement    { makeList("if", 'k', lineCount);  makeList("else", 'k', lineCount); makeList("(", 'p', lineCount);   makeList(")", 'p', lineCount);    }  | SWITCH '(' expression ')' statement   { makeList("switch", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  iteration\_statement  : WHILE '(' expression ')' statement    { makeList("while", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | DO statement WHILE '(' expression ')' ';'   { makeList("do", 'k', lineCount); makeList("while", 'k', lineCount); makeList("(", 'p', lineCount);           makeList(")", 'p', lineCount); makeList(";", 'p', lineCount);   }  | FOR '(' expression\_statement expression\_statement ')' statement    { makeList("for", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  | FOR '(' expression\_statement expression\_statement expression ')' statement   { makeList("for", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }  ;  jump\_statement  : GOTO IDENTIFIER ';' { makeList("goto", 'k', lineCount); makeList(";", 'p', lineCount); makeList(tablePtr, 'v', lineCount);}  | CONTINUE ';' { makeList("continue", 'k', lineCount); makeList(";", 'p', lineCount); }  | BREAK ';'  { makeList("break", 'k', lineCount); makeList(";", 'p', lineCount);}  | RETURN ';'  { makeList("return", 'k', lineCount); makeList(";", 'p', lineCount);}  | RETURN expression ';' { makeList("return", 'k', lineCount); makeList(";", 'p', lineCount);}  ;  translation\_unit  : external\_declaration  | translation\_unit external\_declaration  ;  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  ;  %% void yyerror() {  errorFlag=1;  fflush(stdout);  printf("\n%s : %d :Syntax error \n",sourceCode,lineCount); } void main(int argc,char \*\*argv){  if(argc<=1){    printf("Invalid ,Expected Format : ./a.out <\"sourceCode\"> \n");  return 0;  }    yyin=fopen(argv[1],"r");  sourceCode=(char \*)malloc(strlen(argv[1])\*sizeof(char));  sourceCode=argv[1];  yyparse();    if(nestedCommentCount!=0){  errorFlag=1;     printf("%s : %d : Comment Does Not End\n",sourceCode,lineCount);       }  if(commentFlag==1){  errorFlag=1;  printf("%s : %d : Nested Comment\n",sourceCode,lineCount);     }   if(!errorFlag){    printf("\n\n\t\t%s Parsing Completed\n\n",sourceCode);   FILE \*writeFile=fopen("Output.txt","w"); fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");  fprintf(writeFile,"\n\t\t\t\t\t\t\t\tPARSED TABLE\n");  fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");    fprintf(writeFile,"\t\tIdentifier \t\t\t\tToken\t\t\t\t\t\t Line Number\n");   fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");  for(tokenList \*ptr=parsedPtr;ptr!=NULL;ptr=ptr->next){   fprintf(writeFile,"\n%20s%40s%60s",ptr->token,ptr->type,ptr->line);  }   fprintf(writeFile,"\n \n \n");  fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");  fprintf(writeFile,"\n\t\t\t\t\t\t\t\tSYMBOL TABLE\n");  fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");    fprintf(writeFile,"\tIdentifier \t\t\t\t\tType\t\t\t\t\t\t Line Number\n");   fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");  for(tokenList \*ptr=symbolPtr;ptr!=NULL;ptr=ptr->next){   fprintf(writeFile,"\n%20s%40s%50s",ptr->token,ptr->type,ptr->line);  }   fprintf(writeFile,"\n \n \n");  fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");  fprintf(writeFile,"\n\t\t\t\t\t\t\t\tCONSTANT TABLE\n");  fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");    fprintf(writeFile,"\t\tConstant Value \t\t\t\t\t Line Number\n");   fprintf(writeFile,"---------------------------------------------------------------------------------------------------------------------------------------------\n");  for(tokenList \*ptr=constantPtr;ptr!=NULL;ptr=ptr->next){   fprintf(writeFile,"\n%30s%50s",ptr->token,ptr->line);  }  fclose(writeFile);  } printf("\n\n");  }  void makeList(char \*tokenName,char tokenType, int tokenLine) {  char line[39],lineBuffer[19];     snprintf(lineBuffer, 19, "%d", tokenLine);  strcpy(line," ");  strcat(line,lineBuffer);  char type[20];  switch(tokenType)  {  case 'c':  strcpy(type,"Constant");  break;  case 'v':  strcpy(type,"Identifier");  break;  case 'p':  strcpy(type,"Punctuator");  break;  case 'o':  strcpy(type,"Operator");  break;  case 'k':  strcpy(type,"Keyword");  break;  case 's':  strcpy(type,"String Literal");  break;  case 'd':  strcpy(type,"Preprocessor Statement");  break;  }  for(tokenList \*p=parsedPtr;p!=NULL;p=p->next)   if(strcmp(p->token,tokenName)==0){        strcat(p->line,line);        goto xx;      }  tokenList \*temp=(tokenList \*)malloc(sizeof(tokenList));  temp->token=(char \*)malloc(strlen(tokenName)+1);  strcpy(temp->token,tokenName);  strcpy(temp->type,type);     strcpy(temp->line,line);     temp->next=NULL;          tokenList \*p=parsedPtr;     if(p==NULL){          parsedPtr=temp;     }     else{     while(p->next!=NULL){     p=p->next;     }     p->next=temp;     }          xx:  if(tokenType == 'c')  {          for(tokenList \*p=constantPtr;p!=NULL;p=p->next)   if(strcmp(p->token,tokenName)==0){        strcat(p->line,line);        return;      }  tokenList \*temp=(tokenList \*)malloc(sizeof(tokenList));  temp->token=(char \*)malloc(strlen(tokenName)+1);  strcpy(temp->token,tokenName);  strcpy(temp->type,type);     strcpy(temp->line,line);     temp->next=NULL;          tokenList \*p=constantPtr;     if(p==NULL){     constantPtr=temp;     }     else{     while(p->next!=NULL){     p=p->next;     }     p->next=temp;     }         }  if(tokenType=='v')  {     for(tokenList \*p=symbolPtr;p!=NULL;p=p->next)   if(strcmp(p->token,tokenName)==0){        strcat(p->line,line);        return;      }  tokenList \*temp=(tokenList \*)malloc(sizeof(tokenList));  temp->token=(char \*)malloc(strlen(tokenName)+1);  strcpy(temp->token,tokenName);  switch(typeBuffer){  case 'i': strcpy(temp->type,"INT"); break;  case 'f': strcpy(temp->type,"FLOAT");break;  case 'v' :strcpy(temp->type,"VOID");break;  case 'c': strcpy(temp->type,"CHAR");break;    }       strcpy(temp->line,line);     temp->next=NULL;     tokenList \*p=symbolPtr;     if(p==NULL){          symbolPtr=temp;     }     else{     while(p->next!=NULL){     p=p->next;     }     p->next=temp;     }  } }     %{ *#include <stdio.h>* *#include "y.tab.h"* int lineCount=1; int nestedCommentCount=0; int commentFlag=0; char \*tablePtr; void addToken(char\*); %}  digit [0-9] letter [a-zA-Z\_] hex [a-fA-F0-9] E [Ee][+-]?{digit}+ FS (f|F|l|letter) IS (u|U|l|letter)\*  singlelineComment (\/\/.\*) multilineCommentStart (\/\\*) multilineCommentEnd (\\*\/) %x DETECT\_COMMENT  %%   {singlelineComment}      { }  {multilineCommentStart}            { BEGIN(DETECT\_COMMENT);    nestedCommentCount++;   }   <DETECT\_COMMENT>{multilineCommentStart} { nestedCommentCount++;    if(nestedCommentCount>1)    commentFlag = 1;  }   <DETECT\_COMMENT>{multilineCommentEnd} { BEGIN(INITIAL);   if(nestedCommentCount>0)                                 nestedCommentCount--;                             if(nestedCommentCount==0)                             BEGIN(INITIAL);                     }                      <DETECT\_COMMENT>\n {lineCount++;} <DETECT\_COMMENT>. {}     "auto" {  return(AUTO); } "break" {  return(BREAK); } "case" {  return(CASE); } "char" {  return(CHAR); } "const" {  return(CONST); } "continue" {  return(CONTINUE); } "default" {  return(DEFAULT); } "do" {  return(DO); } "double" {  return(DOUBLE); } "else" {  return(ELSE); } "enum" {  return(ENUM); } "extern" {  return(EXTERN); } "float" {  return(FLOAT); } "for" {  return(FOR); } "goto" {  return(GOTO); } "if" {  return(IF); } "int" {  return(INT); } "long" {  return(LONG); } "register" {  return(REGISTER); } "return" {  return(RETURN); } "short" {  return(SHORT); } "signed" {  return(SIGNED); } "sizeof" {  return(SIZEOF); } "static" {  return(STATIC); } "struct" {  return(STRUCT); } "switch" {  return(SWITCH); } "typedef" {  return(TYPEDEF); } "union" {  return(UNION); } "unsigned" {  return(UNSIGNED); } "void" {  return(VOID); } "volatile" {  return(VOLATILE); } "while" {  return(WHILE); }  {letter}({letter}|{digit})\* {  addToken(yytext); return(IDENTIFIER); } {letter}?\"(\\.|[^\\"])\*\" {  addToken(yytext); return(STRING\_LITERAL); }  0[xX]{hex}+{IS}? {  addToken(yytext); return(CONSTANT); } 0{digit}+{IS}? {  addToken(yytext); return(CONSTANT); } {hex}+{IS}? {  addToken(yytext); return(CONSTANT); } {letter}?'(\\.|[^\\'])+' {  addToken(yytext); return(CONSTANT); } {digit}+{E}{FS}? {  addToken(yytext); return(CONSTANT); } {digit}\*"."{digit}+({E})?{FS}? {  addToken(yytext); return(CONSTANT); } {digit}+"."{digit}\*({E})?{FS}? {  addToken(yytext); return(CONSTANT); }  "..." {  return(ELLIPSIS); } ">>=" {  return(RIGHT\_ASSIGN); } "<<=" {  return(LEFT\_ASSIGN); } "+=" {  return(ADD\_ASSIGN); } "-=" {  return(SUB\_ASSIGN); } "\*=" {  return(MUL\_ASSIGN); } "/=" {  return(DIV\_ASSIGN); } "%=" {  return(MOD\_ASSIGN); } "&=" {  return(AND\_ASSIGN); } "^=" {  return(XOR\_ASSIGN); } "|=" {  return(OR\_ASSIGN); } ">>" {  return(RIGHT\_OP); } "<<" {  return(LEFT\_OP); } "++" {  return(INC\_OP); } "--" {  return(DEC\_OP); } "->" {  return(PTR\_OP); } "&&" {  return(AND\_OP); } "||" {  return(OR\_OP); } "<=" {  return(LE\_OP); } ">=" {  return(GE\_OP); } "==" {  return(EQ\_OP); } "!=" {  return(NE\_OP); } ";" {  return(';'); } ("{"|"<%") {  makeList("{",'p',lineCount); return('{'); } ("}"|"%>") {  makeList("}",'p',lineCount); return('}'); } "," {  return(','); } ":" {  return(':'); } "=" {  return('='); } "(" {  return('('); } ")" {  return(')'); } ("["|"<:") {  return('['); } ("]"|":>") {  return(']'); } "." {  return('.'); } "&" {  return('&'); } "!" {  return('!'); } "~" {  return('~'); } "-" {  return('-'); } "+" {  return('+'); } "\*" {  return('\*'); } "/" {  return('/'); } "%" {  return('%'); } "<" {  return('<'); } ">" {  return('>'); } "^" {  return('^'); } "|" {  return('|'); } "?" {  return('?'); }   "*#include"(.)\*"\n" { lineCount++; }* "#define"(.)\*"\n" { lineCount++; }  [ ] {} [\t\v\f] {} [\n] { lineCount++; } . { }  %% yywrap() {  return(1); }  void addToken(char \*yytext) {   int len = strlen(yytext);   tablePtr = (char\*)malloc((len+1)\*sizeof(char));   strcpy(tablePtr, yytext); } |

Explanation

The Yacc program specifies productions for the following:

* Looping construct: while, for, do-while
* Data types: (signed/unsigned) int, float
* Arithmetic and Relational Operators
* Data structure: Arrays
* User defined functions
* Keywords of C language
* Single and Multi-line comments
* Identifiers and Constant errors
* Selection statement: (nested) if-else

The productions for most of them are straight-forward. A few important ones are:

compound\_statement

: '{' '}'

| '{' statement\_list '}'

| '{' declaration\_list '}'

| '{' declaration\_list statement\_list '}'

;

selection\_statement

: IF '(' expression ')' statement %prec NO\_ELSE

| IF '(' expression ')' statement ELSE 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

: CONTINUE ';'

| BREAK ';'

| RETURN ';'

| RETURN expression ';'

;

statement

: compound\_statement

| expression\_statement

| selection\_statement

| iteration\_statement

| jump\_statement

;

After parsing, if there are errors then the line numbers of those errors are displayed along with a ‘parsing failed’ message on the terminal. Otherwise, a ‘parsing complete’ message is displayed on the console. When a grammar is not carefully thought out, the parser generated from the grammar may face two kinds of dilemmas.

**Shift-Reduce Conflict:** Enough terms have been read and a grammar rule can be recognized according to the accumulated terms. In this situation, the parser can make a reduction. If, however, there is also another grammar rule which calls for more terms to be accumulated and the look ahead token is just what the second grammar rule expected. In this situation, the parser may also make a shift operation. This dilemma faced by the parser is called the Shift/Reduce Conflict.

**Reduce-Reduce Conflict:** Enough terms have been read and two grammar rules are recognized based on the accumulated terms. If the parser then decides to make a reduction, should it reduces the accumulated terms according to the first or second grammar rules? This type of difficulty faced by the parser is called the Reduce/Reduce Conflict.

Test Cases

* Without errors

**TEST CASE 1**: output for testcase containing a function and print statement

**TEST CASE 2**: output for testcase containing do while loop

**TEST CASE 3**: output for testcase containing nested for loop

**TEST CASE 4**: output for testcase containing a structure and string

* With errors

**TEST CASE 5**: output displays error :-double quotes

**TEST CASE 6**: output displays error :- parenthesis, semicolon

**TEST CASE 7**: output displays error :- unmatched parenthesis

**TEST CASE 8**: output displays error :– missing multiple colon

**Code and screenshots**

**Without errors:**

|  |  |
| --- | --- |
| 1)  #include<stdio.h>  int f(int a)  {  if(a==1)return 1;  if(a==0)return 1;  return f(a-1)+f(a-2);  }  void main()  {  if(f(6) != 13)  printf("FAIL\n");  else  printf("PASS\n");  }  2) |  |
| //without error: do while loop  #include<stdio.h>  #include<conio.h>  int main()  {  int num=1; //initializing the variable  do //do-while loop  {  printf("%d\n",2\*num);  num++; //incrementing operation  }while(num<=10);  return 0;  }  3) |  |
| //without error:nested for loop  #include <stdio.h>  int main() {  int i, j;  int table = 2;  int max = 5;  for (i = 1; i <= table; i++) { // outer loop  for (j = 0; j <= max; j++) { // inner loop  printf("%d x %d = %d\n", i, j, i\*j);  }  printf("\n"); /\* blank line between tables \*/  }}  4) |  |
| #include<stdio.h>  struct st{  int a;  };    void main()  {  char s[30];  s= "nitk surathkal";  printf("\nString is:",s);  return;  } |  |

With errors:

1)

|  |  |
| --- | --- |
| #include<stdio.h>  void main()  {  char s[30];  s= "nitk surathkal;  printf("\nString is:",s);  return;  }  2) |  |
| // with error :nested if else parenthesis,semicolon  #include <stdio.h>  int main()  {  int var1, var2;  printf("Input the value of var1:");  scanf("%d", &var1);  printf("Input the value of var2:");  scanf("%d",&var2);  if (var1 != var2)  {  printf("var1 is not equal to var2\n");  //Nested if else  if (var1 > var2)  {  printf("var1 is greater than var2\n");  }  else  {  printf("var2 is greater than var1\n");  }  }  else  {  printf("var1 is equal to var2\n")  }  return 0;  3) |  |
| //with error : unmatched parenthesis    #include<stdio.h>  // declaring the recursive function  int isPrime(int, int);    int main()  {  int num, prime;  printf("Enter a positive number to check if Prime: ");  scanf("%d", &num);  prime = isPrime(num, num/2);  if(prime == 1)  {  printf("\n\n%d is a prime number\n\n", num);  }  else  {  printf("\n\n%d is a Composite number\n\n", num);  }  if eif e return 0;  }    4)  // function definition  int isPrime(int n, int i)  {  if(i == 1)  return 1; // return statement terminates the recursive funtion  else  {  if(n%i == 0)  return 0;  else  isPrime(n, i-1); // recursive call not using return statement  } |  |
| 5)  //with error - missing multiple semicolon  #include<stdio.h>  #define x 3  int main()  {  //int c=4;  int b=5  /\* multiline  comment\*/  printf("%d",b);  printf("this is for checking git commit");  while(b<10)  {  printf("%d", b)  b++;  }  } |  |

Implementation

**HANDLING SHIFT-REDUCE AND REDUCE-REDUCE CONFLICTS**

The yacc command is built with two internal rules for resolving these two ambiguities, sometimes called the disambiguating rules.

1. yacc resolves the Shift/Reduce Conflict in favour of shift operation. In plain English, this just means that it matches the longest input possible.
2. yacc resolves the Reduce/Reduce Conflict in favour of the first grammar rule.

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, it is important to write unambiguous grammar with no conflicts.

**SOLVING DANGLING ELSE PROBLEM**

Whenever it is not possible to associate an ‘else’ to a closest ‘if’ in if-else statements, it gives rise to dangling else problem. In this case, the problem of dangling else occurs here, represented by #:

IF '(' expression ')' statement # ELSE statement

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.

//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

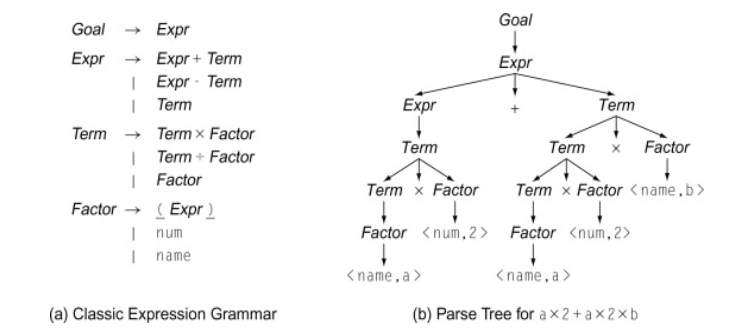
;

%%

Parse Tree

Example Case : a × 2 + a × 2 × b

The parse tree for above expression for the following grammar is shown below:



Conclusion

The lexical analyzer and the syntax analyzer for a subset of C language, which include selection statements, compound statements, iteration statements, jumping statements, user defined functions and primary expressions is generated. It is important to note that conflicts (shift-reduce and reduce-reduce) may occur in case of syntax analyzer if proper care is not taken while specifying the context-free grammar for the language. We should always specify unambiguous grammar for the parser to work properly.

Results and Future Work

We have implemented the parser for only a subset of C language. We were able to successfully parse the tokens recognized by the flex script for C. The output displays the set of identifiers and constants present in the program with their types. The parser generates error messages in case of any syntactical errors in the test program. The future work may include specifying the grammar for more pre-defined functions in C (like string functions, file read and write functions), goto jump statements and enumerations

References

* Compilers Principles, Techniques and Tool by Alfred V.Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
* <http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf> - Lex and Yacc Tutorial by Tom Nieman
* <https://www.programiz.com/c-programming/precedence-associativity> -operators - C Precedence and Associativity
* <http://dinosaur.compilertools.net/yacc/>
* <https://www2.cs.arizona.edu/classes/cs453/fall14/DOCS/tutorial-large.pdf>

**\*\*\*\*\*\*\***