PARSER FOR THE C-LANGUAGE

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA SURATHKAL

Date:
16-9-2020

Submitted To:
P.Santhi Thilagam

Group Members:
Sai Nishanth K R(181CO145)
Srikrishna G Yaji(181CO153)
Shashank D(181CO248)
Tarun S Anur(181CO255)

ABSTRACT

A parser is a compiler component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree.

The main job of a parser for a programming language is to read the source program and discover its structure. Lex and Yacc can generate program fragments that solve this task. The task of discovering the source structure again is decomposed into subtasks:

1. Split the source file into tokens (Lex).
2. Find the hierarchical structure of the program (Yacc).

INDEX

LIST OF FIGURES AND TABLES:

INTRODUCTION

Syntax analysis and Parser

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.

Context Free Grammar

A context-free grammar has four components:

- A set of non-terminals (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as terminal symbols (Σ). Terminals are the basic symbols from which strings are formed.
- A set of productions (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.
- One of the non-terminals is designated as the start symbol (S); from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

Left-most Derivation

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

Right-most Derivation

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree.

YACC (Yet Another Compiler-Compiler)

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine. Lexer can be used to make a simple parser. But it needs making extensive use of the user-defined states.
The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual inputParser for C Language characters, or in terms of higher-level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification. Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

CODE:

Scanner.l (Updated version)

```
%{
   #include<stdio.h>
   #include<string.h>
   #include"y.tab.h"

   struct symbol_table {
      char type[100];
      char name[100];
      char Class[100];
      char value[100];
      int line_number;
      int valid;
   }st[501];

   struct constant_table {
      char type[100];
      char name[100];
      int valid;
   }ct[501];

   void insert_symbol_table_line(char *str1, int line)
   {
      for(int i = 0 ; i < 501 ; i++)
      {
         if(strcmp(st[i].name,str1)==0)
         {
            st[i].line_number = line;
         }
      }
   }

   int hash_function(char *str)
   {
      int value = 0;
      for(int i = 0 ; i < strlen(str) ; i++)
      {
         value = 10*value + (str[i] - 'A');
         value = value % 501;
         while(value < 0)
            value = value + 501;
      }
      return value;
   }
   int lookup_symbolTable(char *str)
   {
      int value = hash_function(str);
```

```c
    if(st[value].valid == 0)
    {
        return 0;
    }
    else if(strcmp(st[value].name,str)==0)
    {
        return 1;
    }
    else
    {
        for(int i = value + 1 ; i!=value ; i = (i+1)%501)
        {
            if(strcmp(st[i].name,str)==0)
            {
                return 1;
            }
        }
        return 0;
    }
}
int lookup_constantTable(char *str)
{
    int value = hash_function(str);
    if(ct[value].valid == 0)
        return 0;
    else if(strcmp(ct[value].name,str)==0)
        return 1;
    else
    {
        for(int i = value + 1 ; i!=value ; i = (i+1)%501)
        {
            if(strcmp(ct[i].name,str)==0)
            {
                return 1;
            }
        }
        return 0;
    }
}
void insert_symbolTable(char *str1, char *str2)
{
    if(lookup_symbolTable(str1))
    {
        return;
    }
    else
    {
        int value = hash_function(str1);
        if(st[value].valid == 0)
        {
            strcpy(st[value].name,str1);
            strcpy(st[value].Class, str2);
            st[value].valid = strlen(str1);
            insert_symbol_table_line(str1, yylineno);
            return;
        }

        int pos = 0;

        for (int i = value + 1 ; i!=value ; i = (i+1)%501)
        {
            if(st[i].valid == 0)
            {
                pos = i;
                break;
            }
        }

        strcpy(st[pos].name,str1);
        strcpy(st[pos].Class,str2);
        st[pos].valid = strlen(str1);
    }
}

void insert_symbol_table_type(char *str1, char *str2)
{
    for(int i = 0 ; i < 501 ; i++)
    {
        if(strcmp(st[i].name,str1)==0)
        {
            strcpy(st[i].type,str2);
        }
    }
}

void insert_symbol_table_value(char *str1, char *str2)
{
    for(int i = 0 ; i < 501 ; i++)
    {
        if(strcmp(st[i].name,str1)==0)
        {
            strcpy(st[i].value,str2);
        }
    }
}


void insert_constantsTable(char *str1, char *str2)
{
    if(lookup_constantTable(str1))
        return;
    else
```

```
        {
            int value = hash_function(str1);
            if(ct[value].valid == 0)
            {
                strcpy(ct[value].name,str1);
                strcpy(ct[value].type,str2);
                ct[value].valid = strlen(str1);
                return;
            }

            int pos = 0;

            for (int i = value + 1 ; i!=value ; i = (i+1)%501)
            {
                if(ct[i].valid == 0)
                {
                    pos = i;
                    break;
                }
            }

            strcpy(ct[pos].name,str1);
            strcpy(ct[pos].type,str2);
            ct[pos].valid = strlen(str1);
        }
    }

    void print_symbol_table()
    {
        printf("%10s | %15s | %10s | %10s | %10s\n","SYMBOL", "CLASS", "TYPE","VALUE", "LINE NO");
        for(int i=0;i<81;i++) {
            printf("-");
        }
        printf("\n");
        for(int i = 0 ; i < 501 ; i++)
        {
            if(st[i].valid == 0)
            {
                continue;
            }
            printf("%10s | %15s | %10s | %10s | %10d\n",st[i].name, st[i].Class, st[i].type, st[i].value, st[i].line_num-
ber);
        }
    }

    void print_constant_table()
    {
        printf("%10s | %15s\n","NAME", "TYPE");
        for(int i=0;i<81;i++) {
            printf("-");
        }
        printf("\n");
        for(int i = 0 ; i < 501 ; i++)
        {
            if(ct[i].valid == 0)
                continue;

            printf("%10s | %15s\n",ct[i].name, ct[i].type);
        }
    }

    int cbracketsopen = 0;
    int cbracketsclose = 0;
    int bbracketsopen = 0;
    int bbracketsclose = 0;
    int fbracketsopen = 0;
    int fbracketsclose = 0;

    char Match_str[20];
    char Match_type[20];
    char curval[20];

%}

identifier [a-zA-Z_][a-zA-Z0-9_]*
numerical_constants (([0-9]*\.[0-9]+)|([0-9]+\.[0-9]*)|([0-9]+))
char_constants  [\t\n ]*((\'[a-zA-Z0-
9]\')|\'\\a\'|\'\\b\'|\'\\e\'|'\\f\'|\'\\n\'|\'\\n\'|\'\\r\'|\'\\t\'|\'\\v\'|\'\\\'|\'\'\'|\'\"\'|\'\?\')
string_constants [\t\n ]*\"(.)*\"
keywords_1 auto|double|int|struct|break|else|long|switch|case|enum|register|typedef|char|extern|re-
turn|union|continue|for|signed
keywords_2 void|do|if|static|while|default|goto|volatile|const|float|short|unsigned
multiline_comment \/\*([^(\*/)]*|(\n)*)\*\/
singleline_comment \/\/(.)*
binary_operators \+|\-|\*|\/|\%
unary_operators \+\+|\-\-
relational_operators \=\=|\!\=|\>|\<|\>\=|\<\=
logical_operators \&\&|\|\||\!
bitwise_operators \&|\||\^|\<\<|\>\>
assignment_operators \+\=|\-\=|\*\=|\/\=|\%\=|\=
special_operators sizeof
special_symbols \[|\]|\{|\}|\(|\)|\,|\;
header_files #(.)*
constants {numerical_constants}|{char_constants}|{string_constants}
operators {unary_operators}|{special_operators}|{logical_operators}|{relational_operators}|{bitwise_opera-
tors}|{binary_operators}|{assignment_operators}

%%


\n {yylineno++;}
[\n\t' ']* {
    for(int i = 0; i < strlen(yytext); i++) {
        if(yytext[i] == '\n') yylineno++;
    }
}
{singleline_comment} ;
{multiline_comment} {
     for(int i = 0; i < strlen(yytext); i++) {
        if(yytext[i] == '\n') yylineno++;
    }
}
{header_files} {printf("%s is a header declaration\n", yytext);}


":"             { return(':'); }
"."

{keywords_1}|{keywords_2} {
    printf("%s is a keyword\n", yytext);
    if(strcmp(yytext, "auto") == 0){
        strcpy(Match_type, yytext);
        insert_symbolTable(yytext, "KEYWORD");
        return(AUTO);
    }
    else if(strcmp(yytext, "double") == 0){
        strcpy(Match_type, yytext);
        insert_symbolTable(yytext, "KEYWORD");
        return(DOUBLE);
    }
    else if(strcmp(yytext, "int") == 0){
        strcpy(Match_type, yytext);
        insert_symbolTable(yytext, "KEYWORD");
        return(INT);
    }
    else if(strcmp(yytext, "struct") == 0){
        strcpy(Match_type, yytext);
        insert_symbolTable(yytext, "KEYWORD");
        return(STRUCT);
    }
    else if(strcmp(yytext, "break") == 0){
        insert_symbolTable(yytext, "KEYWORD");
        return(BREAK);
    }
    else if(strcmp(yytext, "else") == 0){
        insert_symbolTable(yytext, "KEYWORD");
        return(ELSE);
    }
    else if(strcmp(yytext, "long") == 0){
        strcpy(Match_type, yytext);
        insert_symbolTable(yytext, "KEYWORD");
        return(LONG);
    }
```

```
else if(strcmp(yytext, "switch") == 0){
   insert_symbolTable(yytext, "KEYWORD");
   return(SWITCH);
}
else if(strcmp(yytext, "case") == 0){
   insert_symbolTable(yytext, "KEYWORD");
   return(CASE);
}
else if(strcmp(yytext, "enum") == 0){
   insert_symbolTable(yytext, "KEYWORD");
   return(ENUM);
}
else if(strcmp(yytext, "register") == 0){
   strcpy(Match_type, yytext);
   insert_symbolTable(yytext, "KEYWORD");
   return(REG);
}
else if(strcmp(yytext, "typedef") == 0){
   insert_symbolTable(yytext, "KEYWORD");
   return(TYPEDEF);
}
else if(strcmp(yytext, "char") == 0){
   strcpy(Match_type, yytext);
   insert_symbolTable(yytext, "KEYWORD");
   return(CHAR);
}
else if(strcmp(yytext, "extern") == 0){
   strcpy(Match_type, yytext);
   insert_symbolTable(yytext, "KEYWORD");
   return(EXTERN);
}
else if(strcmp(yytext, "return") == 0){
   insert_symbolTable(yytext, "KEYWORD");
   return(RETURN);
}
else if(strcmp(yytext, "union") == 0){
   strcpy(Match_type, yytext);
   insert_symbolTable(yytext, "KEYWORD");
   return(UNION);
}
else if(strcmp(yytext, "continue") == 0){
   insert_symbolTable(yytext, "KEYWORD");
   return(CONTINUE);
}
else if(strcmp(yytext, "for") == 0){
   insert_symbolTable(yytext, "KEYWORD");
   return(FOR);
}
else if(strcmp(yytext, "signed") == 0){
   strcpy(Match_type, yytext);
   insert_symbolTable(yytext, "KEYWORD");
```

```
    return(SIGNED);
  }
  else if(strcmp(yytext, "void") == 0){
    strcpy(Match_type, yytext);
    insert_symbolTable(yytext, "KEYWORD");
    return(VOID);
  }
  else if(strcmp(yytext, "do") == 0){
    insert_symbolTable(yytext, "KEYWORD");
    return(DO);
  }
  else if(strcmp(yytext, "if") == 0){
    insert_symbolTable(yytext, "KEYWORD");
    return(IF);
  }
  else if(strcmp(yytext, "static") == 0){
    insert_symbolTable(yytext, "KEYWORD");
    return(STATIC);
  }
  else if(strcmp(yytext, "while") == 0){
    insert_symbolTable(yytext, "KEYWORD");
    return(WHILE);
  }
  else if(strcmp(yytext, "default") == 0){
    insert_symbolTable(yytext, "KEYWORD");
    return(DEFAULT);
  }
  else if(strcmp(yytext, "goto") == 0){
    insert_symbolTable(yytext, "KEYWORD");
    return(GOTO);
  }
  else if(strcmp(yytext, "volatile") == 0){
    insert_symbolTable(yytext, "KEYWORD");
    return(VOLATILE);
  }
  else if(strcmp(yytext, "const") == 0){
    insert_symbolTable(yytext, "KEYWORD");
    return(CONST);
  }
  else if(strcmp(yytext, "float") == 0){
    insert_symbolTable(yytext, "KEYWORD");
    return(FLOAT);
  }
  else if(strcmp(yytext, "short") == 0){
    insert_symbolTable(yytext, "KEYWORD");
    return(SHORT);
  }
  else if(strcmp(yytext, "unsigned") == 0){
    insert_symbolTable(yytext, "KEYWORD");
    return(UNSIGNED);
  }
  }

({numerical_constants}){identifier} {
  printf("In LineNo: %d, ERROR: Invalid Identifier : %s\n", yylineno, yytext);  exit(1);
}
(\")(\s|{identifier}|{numerical_constants}|{operators})* {
   printf("In LineNo: %d, ERROR: String usage error in %s\n", yylineno, yytext);  exit(1);
}

[\n\t ]*\'(\s|{identifier}|{numerical_constants}|{operators})* {
   printf("In LineNo: %d, ERROR: Character usage error: %s\n", yylineno, yytext);  exit(1);
}

{identifier} {
   printf("%s is a identifier\n", yytext);
   strcpy(Match_str, yytext);
   insert_symbolTable(yytext, "Identifier");
   return(IDENTIFIER);
}

{numerical_constants} {
   printf("%s is a constant\n", yytext);
   strcpy(curval, yytext);
   // insert_constantsTable(yytext, "Constant");
   for(int i = 0; i < strlen(yytext); i++) {
      if(yytext[i] == '\n') yylineno++;
   }
   insert_constantsTable(yytext, "NUMERICAL CONSTANT");
   return(NUM_CONSTANT);
   }

{char_constants} {
   printf("%s is a constant\n", yytext);
   // insert_constantsTable(yytext, "Constant");
   strcpy(curval, yytext);
   for(int i = 0; i < strlen(yytext); i++) {
      if(yytext[i] == '\n') yylineno++;
   }
   insert_constantsTable(yytext, "CHAR CONSTANT");
   return(CHAR_CONSTANT);
   }

{string_constants} {
   printf("%s is a constant\n", yytext);
   // insert_constantsTable(yytext, "Constant");
   strcpy(curval, yytext);
   for(int i = 0; i < strlen(yytext); i++) {
      if(yytext[i] == '\n') yylineno++;
   }
   insert_constantsTable(yytext, "STRING CONSTANT");
   return(STRING_CONSTANT);
   }


{special_symbols} {
   printf("%s is a special symbol\n", yytext);

   if(yytext[0] == ';') { return(';'); }
   else
   if(yytext[0] == ',') { return(','); }
   else
   if(yytext[0] == '{'){
      fbracketsopen++;
      return('{');
   }
   else if(yytext[0] == '}'){
      fbracketsclose++;
      return('}');
   }
   else if(yytext[0] == '('){
      cbracketsopen++;
      return('(');
   }
   else if(yytext[0] == ')'){
      cbracketsclose++;
       return(')');
   }
   else if(yytext[0] == '['){
      bbracketsopen++;
      return('[');
   }
   else if(yytext[0] == ']'){
      bbracketsclose++;
      return(']');
   }
   }


{unary_operators}|{special_operators}|{logical_operators}|{relational_operators}|{bitwise_operators}|{bi-
nary_operators}|{assignment_operators} {
   printf("%s is an operator\n", yytext);
   if(strcmp(yytext, "++") == 0) return increment;
   else if(strcmp(yytext, "--") == 0) return decrement;
   else if(strcmp(yytext, "<<") == 0) return leftshift;
   else if(strcmp(yytext, ">>") == 0) return rightshift;
   else if(strcmp(yytext, "<=") == 0) return lessthanAssignment;
   else if(strcmp(yytext, "<") == 0) return lessthan;
   else if(strcmp(yytext, ">=") == 0) return greaterthanAssignment;
   else if(strcmp(yytext, ">") == 0) return greaterthan;
   else if(strcmp(yytext, "==") == 0) return equality;
   else if(strcmp(yytext, "!=") == 0) return inequality;
   else if(strcmp(yytext, "&&") == 0) return and;
```

```
    else if(strcmp(yytext, "||") == 0) return or;
    else if(strcmp(yytext, "^") == 0) return xor;
    else if(strcmp(yytext, "*=") == 0) return multiplicationAssignment;
    else if(strcmp(yytext, "/=") == 0) return divisionAssignment;
    else if(strcmp(yytext, "%=") == 0) return moduloAssignment;
    else if(strcmp(yytext, "+=") == 0) return additionAssignment;
    else if(strcmp(yytext, "-=") == 0) return subtractionAssignment;
    else if(strcmp(yytext, "<<=") == 0) return leftshiftAssignment;
    else if(strcmp(yytext, ">>=") == 0) return rightshiftAssignment;
    else if(strcmp(yytext, "&=") == 0) return andAssignment;
    else if(strcmp(yytext, "|=") == 0) return orAssignment;
    else if(strcmp(yytext, "&") == 0) return bitAnd;
    else if(strcmp(yytext, "!") == 0) return not;
    else if(strcmp(yytext, "~") == 0) return negation;
    else if(strcmp(yytext, "|") == 0) return bitOr;
    else if(strcmp(yytext, "-") == 0) return subtract;
    else if(strcmp(yytext, "+") == 0) return add;
    else if(strcmp(yytext, "*") == 0) return multiplication;
    else if(strcmp(yytext, "/") == 0) return divide;
    else if(strcmp(yytext, "%") == 0) return modulo;
    else if(strcmp(yytext, "=") == 0) return assignment;
    }

%%



int yywrap(){
    return(1);
}
```

Parser.y

```
%{
   void yyerror(char* s);
   int yylex();
   #include "stdio.h"
   #include "stdlib.h"
   #include "string.h"
   void ins();
   void insV();
   int flag=0;

   extern char Match_str[20];
   extern char Match_type[20];
   extern char curval[20];

%}

%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
%token RETURN MAIN
%token VOID
%token WHILE FOR DO
%token BREAK
%token ENDIF
%token AUTO SWITCH CASE ENUM REG TYPEDEF EXTERN UNION CONTINUE STATIC DEFAULT GOTO VOLA-
TILE CONST IDENTIFIER NUM_CONSTANT CHAR_CONSTANT STRING_CONSTANT


%nonassoc ELSE

%right leftshiftAssignment rightshiftAssignment
%right xorAssignment orAssignment
%right andAssignment moduloAssignment
%right multiplicationAssignment divisionAssignment
%right additionAssignment subtractionAssignment
%right assignment

%left or
%left and
%left bitOr
%left xor
%left bitAnd
%left equality inequality
%left lessthanAssignment lessthan greaterthanAssignment greaterthan
%left leftshift rightshift
%left add subtract
%left multiplication divide modulo

%right SIZEOF
```

```
%right negation not
%left increment decrement


%start program

%%
program
          : declaration_list;

declaration_list
          : declaration D

D
          : declaration_list
          | ;

declaration
          : variable_declaration
          | function_declaration
          | structure_definition;

variable_declaration
          : type_specifier variable_declaration_list ';'
          | structure_declaration;

variable_declaration_list
          : variable_declaration_identifier V;

V
          : ',' variable_declaration_list
          | ;

variable_declaration_identifier
          : IDENTIFIER { ins(); } vdi;

vdi : identifier_array_type | assignment expression ;

identifier_array_type
          : '[' initilization_params
          | ;

initilization_params
          : NUM_CONSTANT ']' initilization
          | ']' string_initilization;

initilization
          : string_initilization
          | array_initialization
          | ;
```

```
type_specifier
          : INT | CHAR | FLOAT | DOUBLE
          | LONG long_grammar
          | SHORT short_grammar
          | UNSIGNED unsigned_grammar
          | SIGNED signed_grammar
          | VOID ;

unsigned_grammar
          : INT | LONG long_grammar | SHORT short_grammar | ;

signed_grammar
          : INT | LONG long_grammar | SHORT short_grammar | ;

long_grammar
          : INT | ;

short_grammar
          : INT | ;

structure_definition
          : STRUCT IDENTIFIER { ins(); } '{' V1  '}' ';';

V1 : variable_declaration V1 | ;

structure_declaration
          : STRUCT IDENTIFIER variable_declaration_list;


function_declaration
          : function_declaration_type function_declaration_param_statement;

function_declaration_type
          : type_specifier IDENTIFIER '('  { ins();};

function_declaration_param_statement
          : params ')' statement;

params
          : parameters_list | ;

parameters_list
          : type_specifier parameters_identifier_list;

parameters_identifier_list
          : param_identifier parameters_identifier_list_breakup;

parameters_identifier_list_breakup
          : ',' parameters_list
          | ;
```

```
param_identifier
          : IDENTIFIER { ins(); } param_identifier_breakup;

param_identifier_breakup
          : '[' ']'
          | ;

statement
          : expression_statment | compound_statement
          | conditional_statements | iterative_statements
          | return_statement | break_statement
          | variable_declaration;

compound_statement
          : '{' statment_list '}' ;

statment_list
          : statement statment_list
          | ;

expression_statment
          : expression ';'
          | ';' ;

conditional_statements
          : IF '(' simple_expression ')' statement conditional_statements_breakup;

conditional_statements_breakup
          : ELSE statement
          | ;

iterative_statements
          : WHILE '(' simple_expression ')' statement
          | FOR '(' expression ';' simple_expression ';' expression ')'
          | DO statement WHILE '(' simple_expression ')' ';';

return_statement
          : RETURN return_statement_breakup;

return_statement_breakup
          : ';'
          | expression ';' ;

break_statement
          : BREAK ';' ;

string_initilization
          : assignment STRING_CONSTANT { insV(); };

array_initialization
          : assignment '{' array_int_declarations '}';
```

```
array_int_declarations
        : NUM_CONSTANT array_int_declarations_breakup;

array_int_declarations_breakup
        : ',' array_int_declarations
        | ;

expression
        : mutable expression_breakup
        | simple_expression ;

expression_breakup
        : assignment expression
        | additionAssignment expression
        | subtractionAssignment expression
        | multiplicationAssignment expression
        | divisionAssignment expression
        | moduloAssignment expression
        | increment
        | decrement ;

simple_expression
        : and_expression simple_expression_breakup;

simple_expression_breakup
        : or and_expression simple_expression_breakup | ;

and_expression
        : unary_relation_expression and_expression_breakup;

and_expression_breakup
        : and unary_relation_expression and_expression_breakup
        | ;

unary_relation_expression
        : not unary_relation_expression
        | regular_expression ;

regular_expression
        : sum_expression regular_expression_breakup;

regular_expression_breakup
        : relational_operators sum_expression
        | ;

relational_operators
        : greaterthanAssignment | lessthanAssignment | greaterthan
        | lessthan | equality | inequality ;

sum_expression
```

```
	: sum_expression sum_operators term
	| term ;

sum_operators
	: add
	| subtract ;

term
	: term MULOP factor
	| factor ;

MULOP
	: multiplication | divide | modulo ;

factor
	: immutable | mutable ;

mutable
	: IDENTIFIER
	| mutable mutable_breakup;

mutable_breakup
	: '[' expression ']'
	| '.' IDENTIFIER;

immutable
	: '(' expression ')'
	| call | constant;

call
	: IDENTIFIER '(' arguments ')';

arguments
	: arguments_list | ;

arguments_list
	: expression A;

A
	: ',' expression A
	| ;

constant
	: NUM_CONSTANT  { insV(); }
	| STRING_CONSTANT   { insV(); }
	| CHAR_CONSTANT{ insV(); };

%%

extern FILE *yyin;
extern int yylineno;
extern char *yytext;
extern int cbracketsopen;
extern int cbracketsclose;
extern int bbracketsopen;
extern int bbracketsclose;
extern int fbracketsopen;
extern int fbracketsclose;
void insert_symbol_table_type(char *,char *);
void insert_symbol_table_value(char *, char *);
void insert_constantsTable(char *, char *);
void print_constant_table();
void print_symbol_table();

int main(int argc , char **argv)
{
   yyin = fopen(argv[1], "r");
   yyparse();
   if((bbracketsopen-bbracketsclose)){
      printf("ERROR: brackets error [\n");
      flag = 1;
   }
   if((fbracketsopen-fbracketsclose)){
      printf("ERROR: brackets error {\n");
      flag = 1;
   }
   if((cbracketsopen-cbracketsclose)){
      printf("ERROR: brackets error (\n");
      flag = 1;
   }


   if(flag == 0)
   {
      printf("Status: Parsing Complete - Valid\n");
      printf("SYMBOL TABLE\n");
      printf("%30s %s\n", " ", "------------");
      print_symbol_table();
      printf("\n\nCONSTANT TABLE\n");
      printf("%30s %s\n", " ", "--------------");
      print_constant_table();
   }
}

void yyerror(char *s)
{
   puts("========================================================================");
   printf("Parsing Failed at line no: %d\n", yylineno);
   printf("Error: %s\n", yytext);
   flag=1;
}
```

```
void ins()
{
   insert_symbol_table_type(Match_str,Match_type);
}

void insV()
{
   insert_symbol_table_value(Match_str,curval);
}
```

EXPLANATION:

Structure of a YACC source program

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

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

THE DECLARATION SECTION:

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

RULES SECTION:

In this section production rules for entire C language is written. The rules are written in such a way that there is no left recursion and the grammar is also deterministic.

Non-deterministic grammar was converted to deterministic by applying left factoring. This was done so that grammar is for LL(1) parser.

A rule has the form:

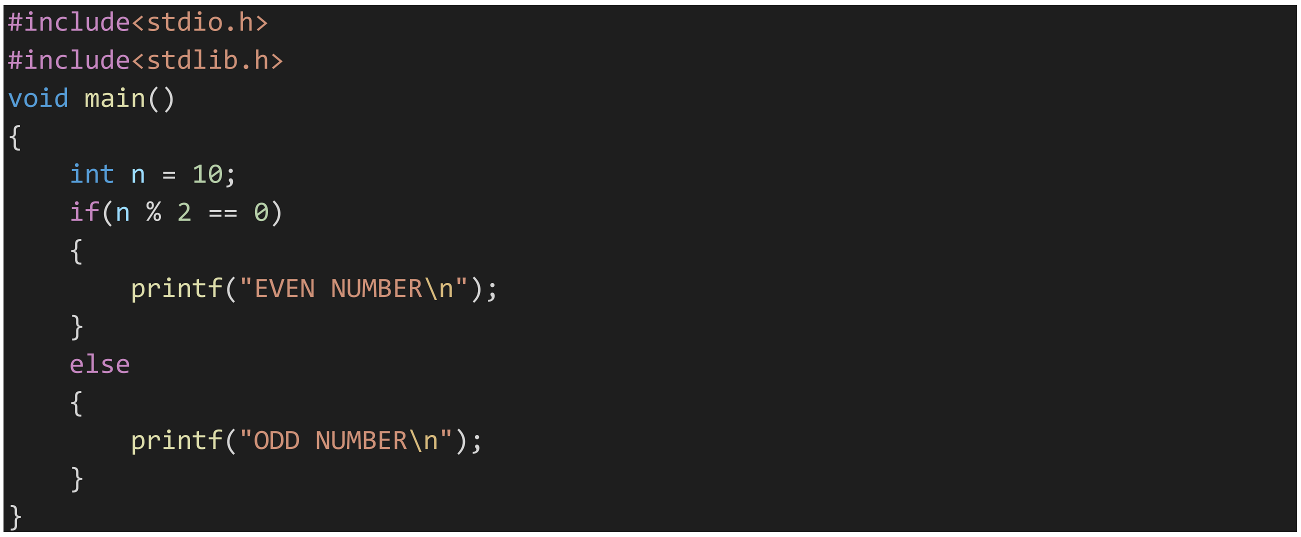
```
nonterminal : sentential form
          | sentential form
          ................
          | sentential form
          ;
```

USER ROUTINES SECTION:

In this section, the user defines some functions that is required for the parse table generation. The parse table is printed out and any parsing errors are identified and printed to the console.

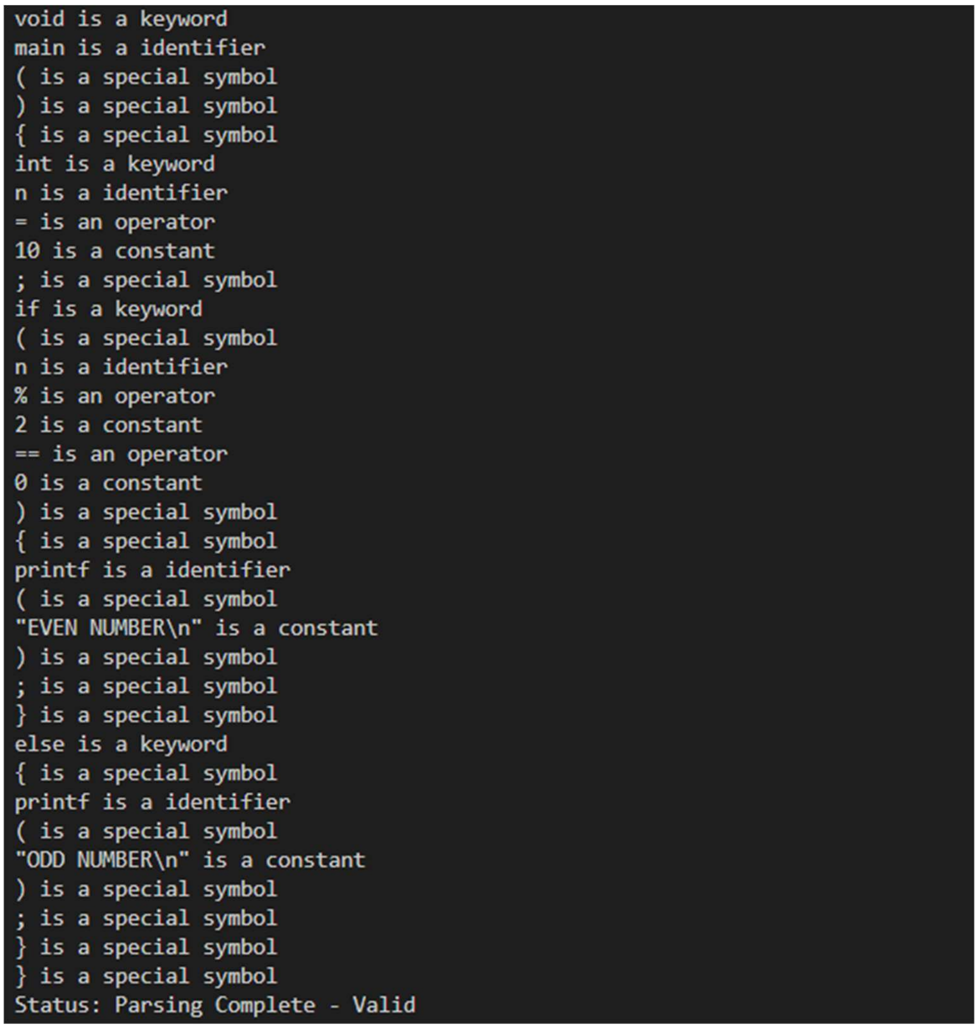
SAMPLE PROGRAMS:

Sample Input 1(Error free):

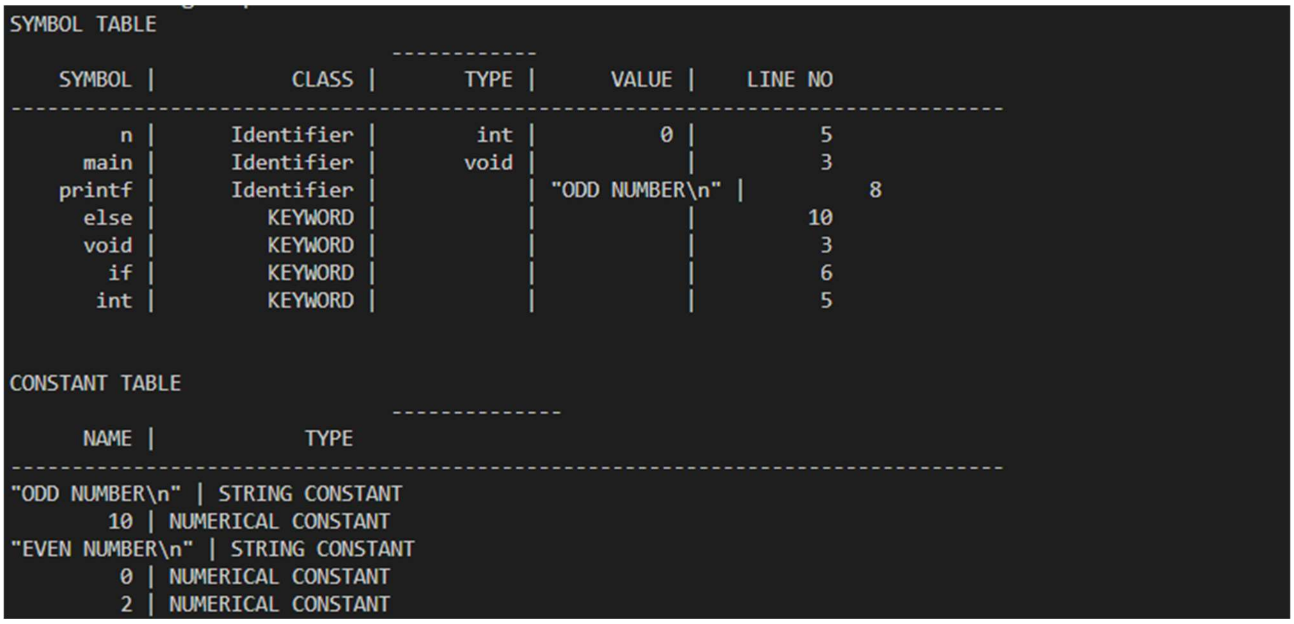


```c
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int n = 10;
    if(n % 2 == 0)
    {
        printf("EVEN NUMBER\n");
    }
    else
    {
        printf("ODD NUMBER\n");
    }
}
```

Sample Output1:

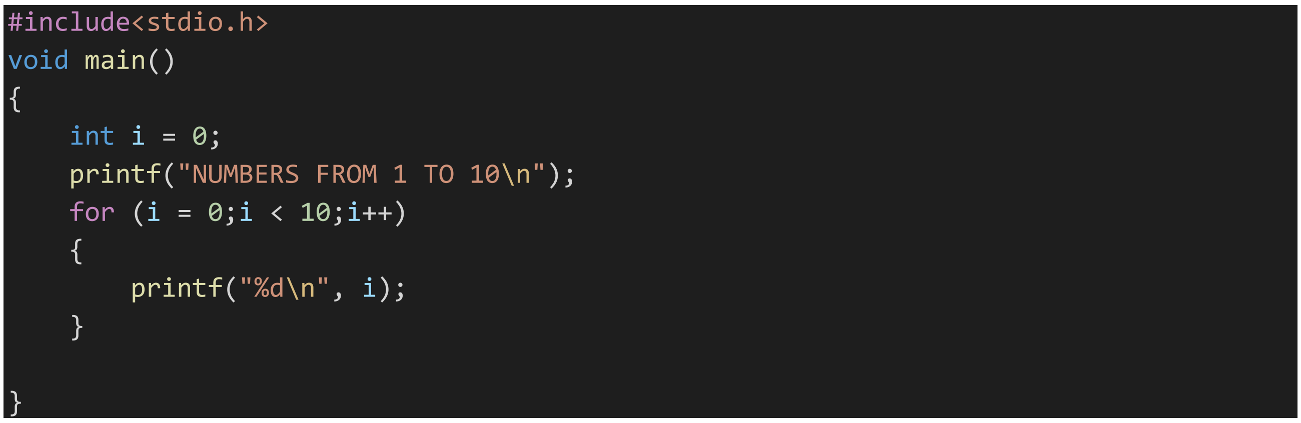


```
void is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
n is a identifier
= is an operator
10 is a constant
; is a special symbol
if is a keyword
( is a special symbol
n is a identifier
% is an operator
2 is a constant
== is an operator
0 is a constant
) is a special symbol
{ is a special symbol
printf is a identifier
( is a special symbol
"EVEN NUMBER\n" is a constant
) is a special symbol
; is a special symbol
} is a special symbol
else is a keyword
{ is a special symbol
printf is a identifier
( is a special symbol
"ODD NUMBER\n" is a constant
) is a special symbol
; is a special symbol
} is a special symbol
} is a special symbol
Status: Parsing Complete - Valid
```



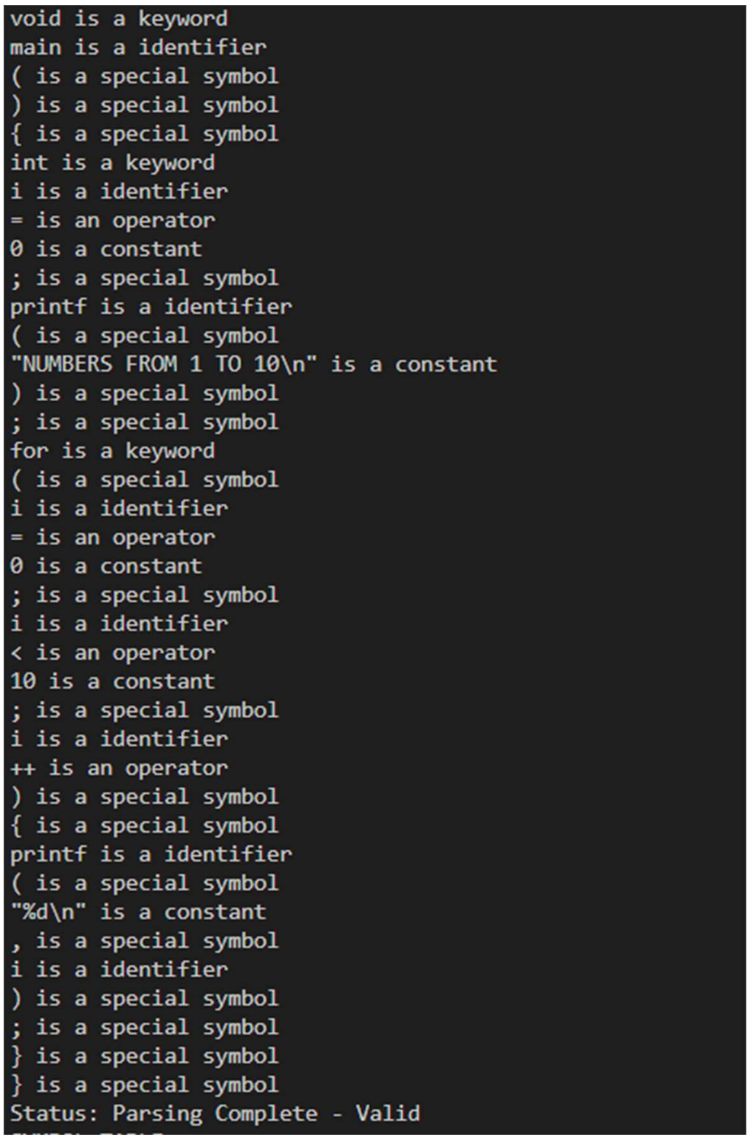
```
SYMBOL TABLE
                                                              ------------
        SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
--------------------------------------------------------------------------------
             n |      Identifier |        int |          0 |          5
          main |      Identifier |       void |            |          3
        printf |      Identifier |            | "ODD NUMBER\n" |          8
          else |         KEYWORD |            |            |         10
          void |         KEYWORD |            |            |          3
            if |         KEYWORD |            |            |          6
           int |         KEYWORD |            |            |          5


CONSTANT TABLE
                                                              --------------
          NAME |            TYPE
--------------------------------------------------------------------------------
"ODD NUMBER\n" | STRING CONSTANT
            10 | NUMERICAL CONSTANT
"EVEN NUMBER\n" | STRING CONSTANT
             0 | NUMERICAL CONSTANT
             2 | NUMERICAL CONSTANT
```

Sample Input 2(Error free):

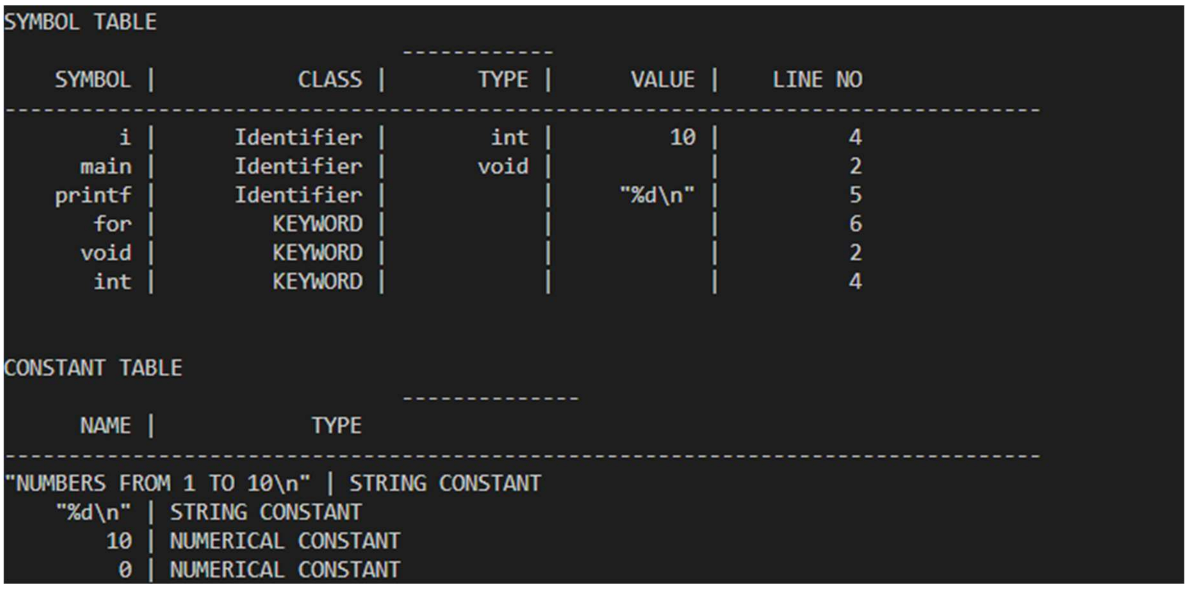


```
#include<stdio.h>
void main()
{
    int i = 0;
    printf("NUMBERS FROM 1 TO 10\n");
    for (i = 0;i < 10;i++)
    {
        printf("%d\n", i);
    }

}
```

Sample Output 2:

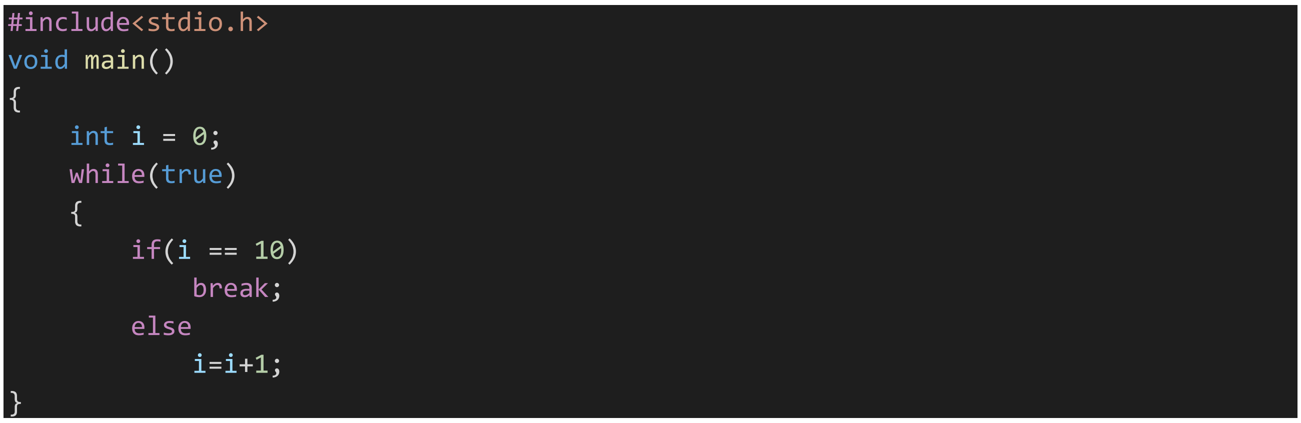


```
void is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
i is a identifier
= is an operator
0 is a constant
; is a special symbol
printf is a identifier
( is a special symbol
"NUMBERS FROM 1 TO 10\n" is a constant
) is a special symbol
; is a special symbol
for is a keyword
( is a special symbol
i is a identifier
= is an operator
0 is a constant
; is a special symbol
i is a identifier
< is an operator
10 is a constant
; is a special symbol
i is a identifier
++ is an operator
) is a special symbol
{ is a special symbol
printf is a identifier
( is a special symbol
"%d\n" is a constant
, is a special symbol
i is a identifier
) is a special symbol
; is a special symbol
} is a special symbol
} is a special symbol
Status: Parsing Complete - Valid
```



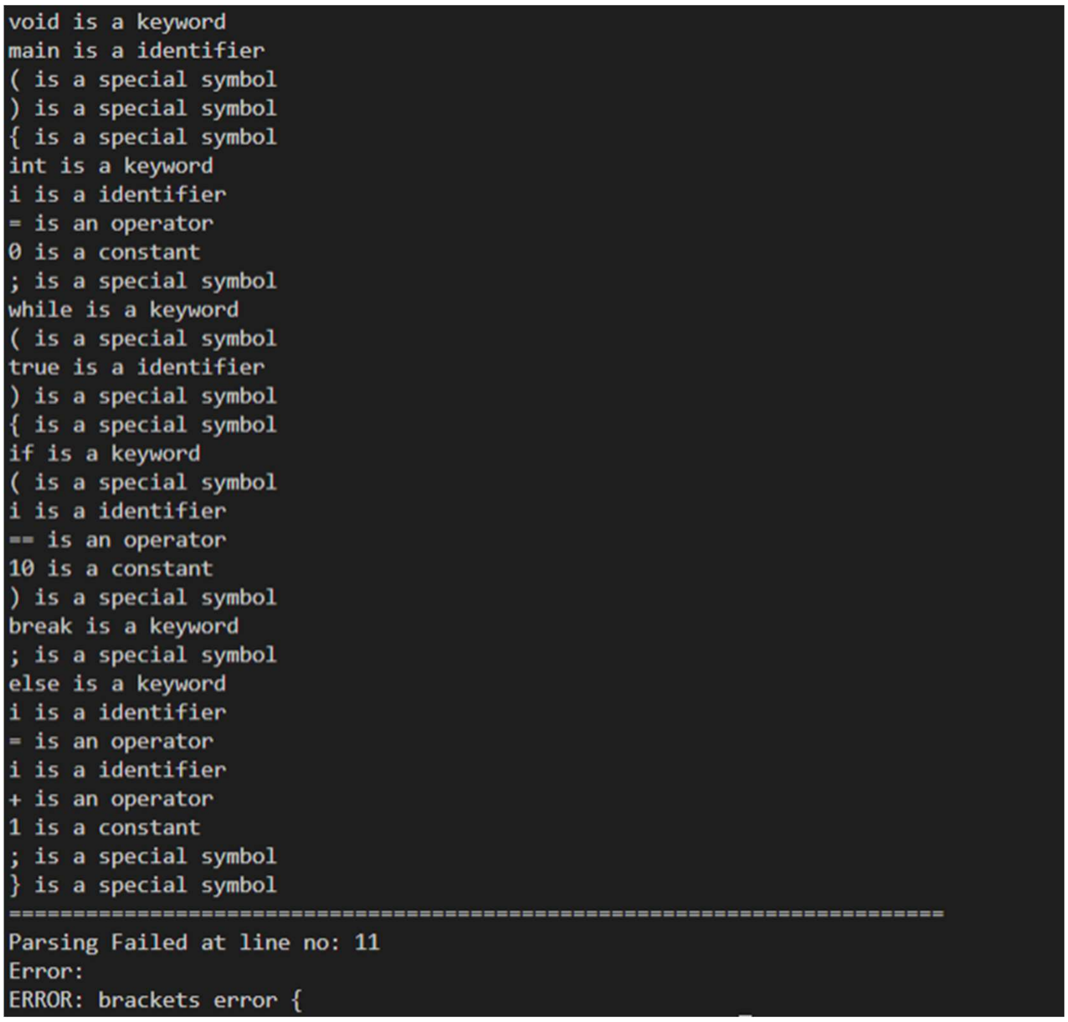
```
SYMBOL TABLE
                                                              -----------
        SYMBOL |            CLASS |        TYPE |       VALUE |     LINE NO
--------------------------------------------------------------------------------
             i |       Identifier |         int |          10 |           4
          main |       Identifier |        void |             |           2
        printf |       Identifier |             |      "%d\n" |           5
           for |          KEYWORD |             |             |           6
          void |          KEYWORD |             |             |           2
           int |          KEYWORD |             |             |           4


CONSTANT TABLE
                                                              -------------
          NAME |             TYPE
--------------------------------------------------------------------------------
"NUMBERS FROM 1 TO 10\n" | STRING CONSTANT
        "%d\n" | STRING CONSTANT
            10 | NUMERICAL CONSTANT
             0 | NUMERICAL CONSTANT
```

Sample Input 3(With Error):



```
#include<stdio.h>
void main()
{
    int i = 0;
    while(true)
    {
        if(i == 10)
            break;
        else
            i=i+1;
}
```

Sample Output 3:

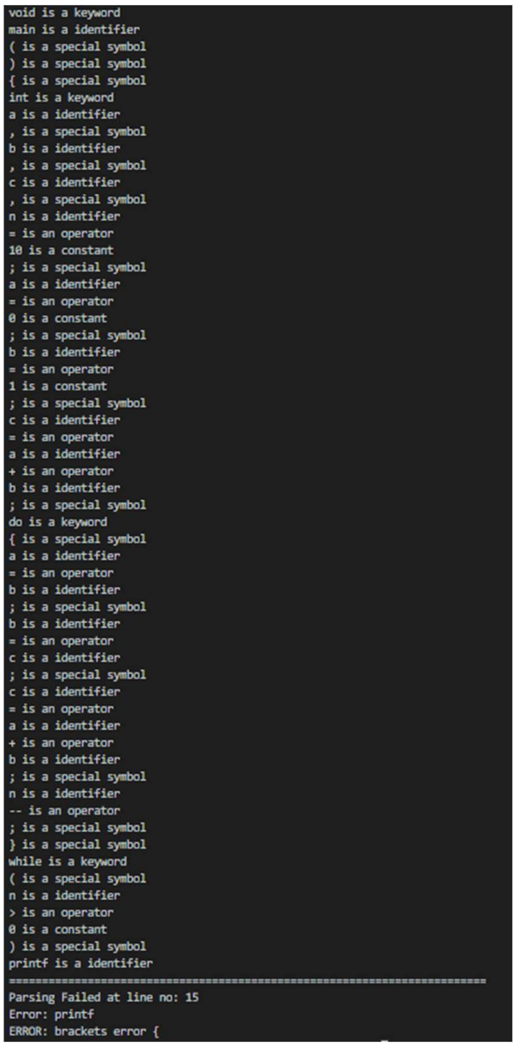


```
void is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
i is a identifier
= is an operator
0 is a constant
; is a special symbol
while is a keyword
( is a special symbol
true is a identifier
) is a special symbol
{ is a special symbol
if is a keyword
( is a special symbol
i is a identifier
== is an operator
10 is a constant
) is a special symbol
break is a keyword
; is a special symbol
else is a keyword
i is a identifier
= is an operator
i is a identifier
+ is an operator
1 is a constant
; is a special symbol
} is a special symbol
=========================================================================
Parsing Failed at line no: 11
Error:
ERROR: brackets error {
```

Sample Input 4:

```
#include<stdio.h>
void main()
{
    int a, b, c, n = 10;
    a = 0;
    b = 1;
    c = a+b;
    do
    {
        a = b;
        b = c;
        c = a+b;
        n--;
    } while (n>0)
    printf("Nth Fibonacci number is %d", c);
}
}
```

Sample Output 4:



```
void is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
a is a identifier
, is a special symbol
b is a identifier
, is a special symbol
c is a identifier
, is a special symbol
n is a identifier
= is an operator
10 is a constant
; is a special symbol
a is a identifier
= is an operator
0 is a constant
; is a special symbol
b is a identifier
= is an operator
1 is a constant
; is a special symbol
c is a identifier
= is an operator
a is a identifier
+ is an operator
b is a identifier
; is a special symbol
do is a keyword
{ is a special symbol
a is a identifier
= is an operator
b is a identifier
; is a special symbol
b is a identifier
= is an operator
c is a identifier
; is a special symbol
c is a identifier
= is an operator
a is a identifier
+ is an operator
b is a identifier
; is a special symbol
n is a identifier
-- is an operator
; is a special symbol
} is a special symbol
while is a keyword
( is a special symbol
n is a identifier
> is an operator
0 is a constant
) is a special symbol
printf is a identifier
================================================================================
Parsing Failed at line no: 15
Error: printf
ERROR: brackets error {
```

ERROR HANDLING:

1. The parser generates errors for every non-recognized grammar.
2. The parser also takes care of the brackets error and prints the line number for each bracket error.

FUTURE DEVELOPMENTS:

1. The unary operations aren’t defined for most of the operators.

BIBILIOGRAPHY:

1. https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm
2. https://en.wikipedia.org/wiki/Parsing