Parser for C Language

Project - 2 (CO351)
7th February 2019

Submitted to :
Dr. P. Santhi Thilagam
Faculty, Dept of Computer Science and Engineering
National Institute of Technology Karnataka, Surathkal

Group members :
Ayush Kumar - 16CO208
Gauri Baraskar - 16CO209
Sanjana Krishnam - 16CO139

Abstract

Objective

The main goal of this project is to design a mini compiler for a subset of the C language as part of the Compiler Design Lab(CO351) course. The compiler is to be built in four phases finishing at the Intermediate Code Generation Phase. The subset of the C language chosen is to include certain data types, constructs and functions as mentioned in the specifications below. The implementation will be carried out with LEX and YACC.

Phases of the project

- Implementation of Scanner/Lexical Analyser
- Implementation of Parser
- Implementation of Semantic Checker for C language
- Intermediate Code generation for C language

Mini-compiler Specifications

The compiler is going to support the following cases :

1. **Keywords** - int, break, continue, else, for, void, goto, char, do, if, return, while
2. **Identifiers** - identified by the regular expression (_ |{letter})({letter}|{digit}| _){0,31}
3. **Comments** - single line comments (specified with // or /* … */), multi-line comments (specified with /* … */)
4. **Strings** - can identify strings mentioned in double quotes
5. **Preprocessor directives** - can identify filenames (stdio.h) after #include
6. **Data types** - int,char (supports comma declaration)
    - Syntax:
        - int a=5;
        - Char newchar = ‘a’
7. **Arrays** - int A[n]
    - Syntax:
        - int A[3]={1,2,3};
8. **Punctuators** - [ ] , <> , {} , , , : , = , ; , # , ” ” ,’ ’
9. **Operators** - arithmetic (+, - , * , /) ,increment(++) and decrement(--), assignment (=)
10. **Condition** - if else
    - Syntax:
        - if (condition == true){
            //code
            }
            else{
            //code
            }

11. **Loops -**
    - Syntax
        - while(condition){
          //code
          }
        - for(initialization;condition;increment/decrement){
          //code
          }
        - while(condition){
          //code
          }
        - do{
          //code
          }while(condition);

    The loop control structures that are supported are break,continue and goto.

12. **Functions-**
    Void functions with no return type and a single parameter will be implemented
    - Syntax
        - void sample_function(int a){
          //code
          }

The mini compiler will be implemented in a straightforward fashion , using Lex and YACC tools starting off with the Scanner as the first module. If time permits we will add more features to cover a larger subset of the C language.

Contents

List of Figures and Tables

Introduction

Parser

A parser is a compiler or interpreter 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.

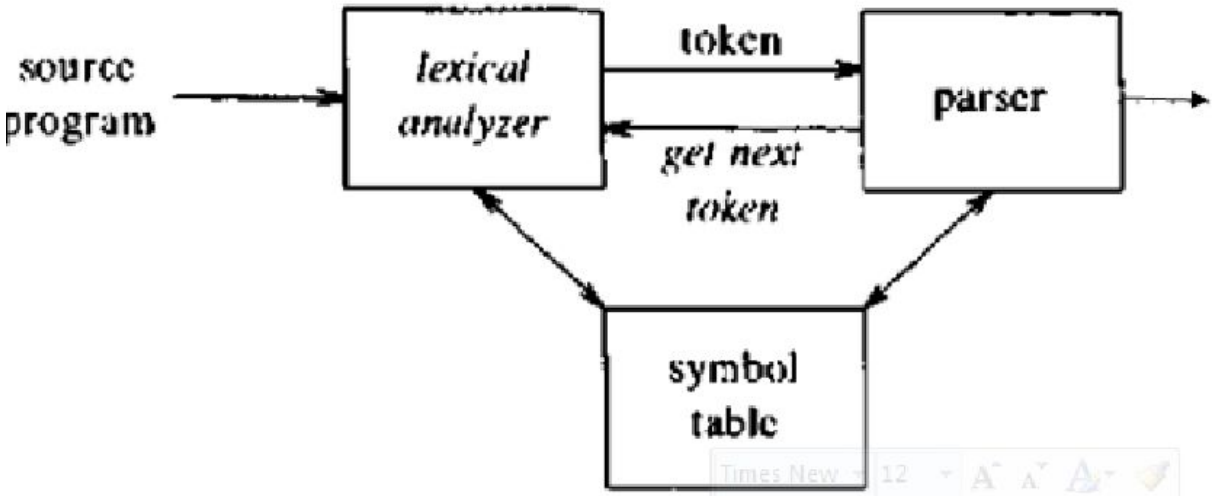


Illustration of Parser's interaction with lexical analyser

In the syntax analysis phase the parser verifies whether or not the tokens generated by the lexical analyser are grouped according to the syntactic rules of the language.The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

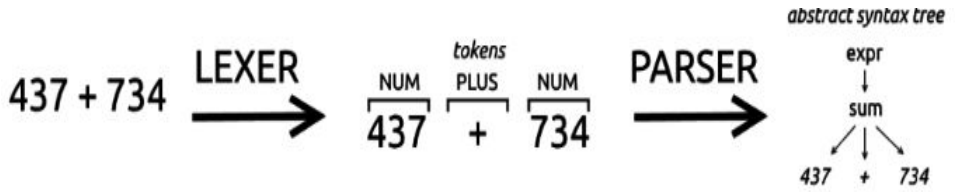
An overview of parsers

A lexer and a parser work in sequence: the lexer scans the input and produces the matching tokens, the parser then scans the tokens and produces the parsing result.

An example : Suppose we have the following statement as part of input

```
437 + 734
```

This is how the above input will get parsed.



Based on the algorithm and design philosophy parsers can be implemented in several ways.
Some parsers don't use lexical analysers or scanners, they are called scannerless parsers.

Some other principles which are crucial to parser design are:

1. Using Leftmost Derivations or Rightmost derivations
2. The core algorithm used , reduction or predictive parsers

Another classification is based on how parsers process the program, they can either process the program from top to bottom (top -down parsers) or they can process it from bottom to top (bottom -up).

Yacc Script

Yacc stands for 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.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

The structure of our Yacc script is given below; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

Definition section

%%

Rules section

%%

C code section

The definition section is used to define any parameters for the C program, any header files to be included and global variable declarations. We also define all parameters related to the parser here, specifications like using Leftmost derivations

or Rightmost derivations, precedence and left right associativity are declared here, data types and tokens which will be used by the lexical analyser are also declared in this stage.

The Rules section contains the entire grammar which is used for deciding if the input text is legally correct according to the specifications of the language. Yacc uses this rules for reducing the token stream received from the scanner, all rules are linked to each other from the start state, which is declared in the rules section.

In the C code section the parser is called, and the symbol table and constant tables are initialised in this section. The lex.yy.c file created by the lex script is also included from here which the parser calls. In this section we also define the error function used by the parser to report syntactical errors along with line numbers.

C Program

The parser(yacc script) takes C source files as input for parsing, the files are supplied using command line arguments, the path is given as argument.

For testing the parser the following commands have to be executed

```
lex lexer_parser.l
yacc -d parser.y
gcc y.tab.c -ll -ly
./a.out testcases/test-file-name.c
```

The C programs we have used for testing induce some errors to verify that the parser can recognise them and some files have syntactically correct code , like dangling if else, which will be used to verify the parser works with the correct programs.

Design of Programs

Updated Lexer Code

```
/* The file describes the rules for our lexer */
%{
    #include<stdio.h>
    #include<stdlib.h>
    #include<limits.h>
    int ErrFlag = 0;

    #include "y.tab.h"
%}

/* Declarations */
letter [a-zA-Z]
digit [0-9]
whitespace [ \t\r\f\v]+
identifier (_|{letter})({letter}|{digit}|_)*
hex [0-9a-fA-F]


/* States */
%x PREPROCESSOR
%x MACROPREPROCESSOR
%x COMMENT
%x SLCOMMENT

%%

/* Keywords */

"int"                               {return INT;}
"short"                             {return SHORT;}
"long"                              {return LONG;}
"char"                              {return CHAR;}
"signed"                            {return SIGNED;}
"unsigned"                          {return UNSIGNED;}
"void"                              {return VOID;}
"if"                                {return IF;}
"else"                              {return ELSE;}
"for"                               {return FOR;}
"do"                                {return DO;}
"while"                             {return WHILE;}
"goto"                              {return GOTO;}
"break"                             {return BREAK;}
"continue"                          {return CONTINUE;}
"main"                              {return MAIN;}
"return"                            {return RETURN;}
"struct"                            {return STRUCT_UNI;}
"union"                             {return STRUCT_UNI;}
"case"                              {return CASE;}
```

```
"default"                           {return DEFAULT;}
"switch"                            {return SWITCH;}


/* Constants */

[0][x|X]{hex}+                      {yylval.dval = (int)strtol(yytext, NULL, 16);
InsertEntry(ConstantTable, yytext , yylval.dval,"HEX",yylineno);return HEX_CONSTANT;}
[+/-]?{digit}+                      {yylval.dval = (int)
atoi(yytext);InsertEntry(ConstantTable, yytext , yylval.dval,"INT",yylineno);return
INT_CONSTANT;}
[+/-]?({digit}*)["."]({digit}+)     {yylval.dval =
atof(yytext);InsertEntry(ConstantTable, yytext , yylval.dval,"FLOAT",yylineno);return
DEC_CONSTANT;}
[+/-]?({digit}+)["."]({digit}*)     {yylval.dval =
atof(yytext);InsertEntry(ConstantTable, yytext , yylval.dval,"FLOAT",yylineno);return
DEC_CONSTANT;}

{identifier} {
  if(strlen(yytext) <= 32)
    {
     yylval.tbEntry = InsertEntry(SymbolTable, yytext, INT_MAX , "INT",yylineno);
      return IDENTIFIER;
    }
  else
  {
      printf("Error %d: Identifier too long,must be between 1 to 32 characters\n",
yylineno);
      ErrFlag = 1;
  }
}


{digit}+({letter}|_)+               {printf("Error %d: Illegal identifier format\n",
yylineno); ErrFlag = 1;}
{whitespace}                        ;


/* Preprocessor Directives */

^"#include"                         {BEGIN PREPROCESSOR;}
<PREPROCESSOR>{whitespace}          ;
<PREPROCESSOR>"<"[^<>\n]*">"        {BEGIN INITIAL;}
<PREPROCESSOR>\"[^<>\n]*\"          {BEGIN INITIAL;}
<PREPROCESSOR>"\n"                  { yylineno++; BEGIN INITIAL; ErrFlag=1;}
<PREPROCESSOR>.                     {yyerror("Improper Header");}


/* Macropreprocessor Directives */

^"#define"                                        {BEGIN MACROPREPROCESSOR;}
<MACROPREPROCESSOR>{whitespace}                   ;
<MACROPREPROCESSOR>({letter})({letter}|{digit})*  {BEGIN INITIAL;}
```

```
<MACROPREPROCESSOR>\n                          {yylineno++; BEGIN INITIAL;}
<MACROPREPROCESSOR>.                           {BEGIN INITIAL;ErrFlag=1;}


/* Comments */

"/*"                                {BEGIN COMMENT;}
<COMMENT>.|{whitespace}             ;
<COMMENT>\n                         {yylineno++;}
<COMMENT>"*/"                       {BEGIN INITIAL;}
<COMMENT>"/*"                       {yyerror("Improper Comment");yyterminate();}
<COMMENT><<EOF>>                    {yyerror("Improper Comment");yyterminate();}
"//"                                {BEGIN SLCOMMENT;}
<SLCOMMENT>.                        ;
<SLCOMMENT>\n                       {yylineno++; BEGIN INITIAL;}


/* Operators */

"+"                                 {return ADD;}
"-"                                 {return SUBTRACT;}
"*"                                 {return MULTIPLY;}
"/"                                 {return DIVIDE;}
"%"                                 {return MOD;}
"="                                 {return ASSIGN;}
"--"                                {return DECREMENT;}
"++"                                {return INCREMENT;}

"+="                                {return ADD_ASSIGN;}
"-="                                {return SUB_ASSIGN;}
"*="                                {return MUL_ASSIGN;}
"/="                                {return DIV_ASSIGN;}
"%="                                {return MOD_ASSIGN;}

">"                                 {return GREATER_THAN;}
"<"                                 {return LESSER_THAN;}
">="                                {return GREATER_EQ;}
"<="                                {return LESS_EQ;}
"=="                                {return EQUAL;}

"||"                                {return LG_OR;}
"&&"                                {return LG_AND;}
"!"                                 {return NOT;}
"!="                                {return NOT_EQ;}


 /* Strings and Characters */

\"[^\"\n]*$                         {ErrFlag=1; yyterminate();}
\"[^\"\n]*\" {
  if(yytext[yyleng-2]=='\\') {
    yyless(yyleng-1);
    yymore();
```

```
  }
  else
  {
    InsertEntry(ConstantTable,yytext,yylval.dval,"CHAR",yylineno);
    return STRING;
  }
}

\'[^\'\n]\'
{InsertEntry(ConstantTable,yytext,yylval.dval,"CHAR",yylineno); return STRING;}


 /* Punctuators */


"\n"                                {yylineno++;}
.                                   {return yytext[0];}

%%
```

In the previous phase the lex script was used to identify tokens and print them on the console and add them to symbol table or constant table. The lexical analyser is supposed to return a stream of tokens to the parser in this stage and so we had to modify the actions taken on matching with a token type. The updated code has been added above, this code uses the same regular expressions used in the last stage,but the lex script just returns the tokens in this case and appends identifiers and constants to the symbol table.

The script still identifies lexical errors and reports them to the parser which then throws a syntax error.

Parser Code

```
/* The file describes the grammar for our parser */

%{
    // Header files
    #include<stdio.h>
         #include<stdlib.h>
         #include "tables.h"

    // Initialising Symbol table and constant table
    entry **SymbolTable = NULL;
    entry **ConstantTable = NULL;

    int yyerror(char *msg);
    char* curr_data_type;
    int yylex(void);
```

```
%}

// Data types of tokens
%union{
        int ival;
        char *str;
        entry *tbEntry;
        double dval;
}

/* Operators */
%token ADD SUBTRACT MULTIPLY DIVIDE ASSIGN ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN
MOD_ASSIGN  MOD
/* Relational Operators */
%token GREATER_THAN LESSER_THAN LESS_EQ GREATER_EQ NOT_EQ EQUAL
/* Keywords */
%token VOID IF ELSE FOR DO WHILE GOTO BREAK CONTINUE RETURN SWITCH CASE DEFAULT
STRUCT_UNI MAIN
/* Data types */
%token INT SHORT LONG CHAR SIGNED UNSIGNED
/* Logical Operators */
%token LG_OR LG_AND NOT
/* Assignment Operators */
%token DECREMENT INCREMENT
/* Constants */
%token <dval> HEX_CONSTANT DEC_CONSTANT INT_CONSTANT
/* String */
%token <str> STRING
/* Identifier */
%token <tbEntry> IDENTIFIER

// Start Symbol of the grammar
%start program

/* Precedence of Operators */
%left ','
%right ASSIGN
%left LG_OR
%left LG_AND
%left EQUAL NOT_EQ
%left LESSER_THAN GREATER_THAN LESS_EQ GREATER_EQ
%left ADD SUBTRACT
%left MULTIPLY DIVIDE MOD
%right NOT


%nonassoc UMINUS UPLUS
%nonassoc IFX
%nonassoc ELSE
%%
    /* Program is made up of declarations */
    program : declarationList;
    // Program can have multiple declarations
```

```
    declarationList : declarationList declaration | declaration;
    // Types of declaration in C
    declaration : varDeclaration | funDeclaration | strucUniDecl;

    // Grammar for struct and union
    strucUniDecl : STRUCT_UNI IDENTIFIER '{' declarationRecur '}' idBlock ';';
    idBlock : idBlock ',' IDENTIFIER | idBlock ',' IDENTIFIER '[' INT_CONSTANT ']' |
IDENTIFIER | IDENTIFIER '[' INT_CONSTANT ']';
    declarationRecur : declarationRecur declaration | declaration ;

    // Variable declarations
    /* Variable declaration can be a list */
    varDeclaration : typeSpecifier varDeclList ';' ;
    // Variables can also be initialised during declaration
    varDeclList : varDeclList ',' varDeclInitialize | varDeclInitialize;
    // Assigment can be through a simple expression or conditional statement
    varDeclInitialize : varDecId | varDecId ASSIGN STRING | varDecId ASSIGN
assignmentExpression ;
    varDecId : IDENTIFIER {$1->data_type = curr_data_type;} | IDENTIFIER '['
INT_CONSTANT ']';
    assignmentExpression : conditionalStmt | unaryExpression assignmentOperator
assignmentExpression;
    assignmentOperator : ASSIGN | ADD_ASSIGN | SUB_ASSIGN
|MUL_ASSIGN|DIV_ASSIGN|MOD_ASSIGN;

    // Types for constants
    const_type : DEC_CONSTANT
               | INT_CONSTANT
               | HEX_CONSTANT
               ;
    // data types
    typeSpecifier : typeSpecifier pointer
                  | INT {curr_data_type = strdup("INT");}
                  | LONG INT
                  | CHAR {curr_data_type = strdup("CHAR");}
                  ;
    // Pointer declaration
    pointer : MULTIPLY pointer | MULTIPLY;

    // Function declaration
    funDeclaration : typeSpecifier IDENTIFIER '(' params ')' compoundStmt |
typeSpecifier MAIN '(' params ')' compoundStmt | noDefDeclare ;
    noDefDeclare : typeSpecifier IDENTIFIER '(' params ')' ';';
    funCall : IDENTIFIER '(' params ')' statement;

    // Rules for parameter list
    params : paramList | ;
    paramList :paramList ',' paramTypeList | paramTypeList ;
    paramTypeList : typeSpecifier paramId;
    paramId : IDENTIFIER | IDENTIFIER '[' ']';

    // Types os statements in C
    statement : labeledStmt | expressionStmt  | compoundStmt  | selectionStmt |
```

```
iterationStmt | jumpStmt | returnStmt | breakStmt | funCall ;

    // Matches label for goto and grammar for switch statement
    labeledStmt : IDENTIFIER ':' statement | CASE conditionalStmt ':' statement |
DEFAULT ':' statement

    // compound statements produces a list of statements with its local declarations
    compoundStmt : '{' localDeclarations statementList '}' ;
    localDeclarations : localDeclarations varDeclaration
                      |  ;
    statementList : statementList statement
                  |  ;
    // Expressions
    expressionStmt : expression ';' | ';' ;
    selectionStmt : IF '(' simpleExpression ')' statement %prec IFX
                  | IF '(' simpleExpression ')' statement ELSE statement
                  | SWITCH '(' simpleExpression ')' statement;

    iterationStmt : WHILE '(' simpleExpression ')' statement
                  | DO statement WHILE '(' expression ')' ';'
                  | FOR '(' optExpression ';' optExpression ';' optExpression ')'
statement;
    // Optional expressions in case of for
    optExpression : expression | ;

    jumpStmt : GOTO IDENTIFIER ';' | CONTINUE ';' ;
    returnStmt : RETURN ';'
               | RETURN expression ;
    breakStmt : BREAK ';' ;

    conditionalStmt : simpleExpression '?' expression ':' conditionalStmt  |
simpleExpression;


    // All arithmetic expressions
    expression : IDENTIFIER assignmentOperator expression
               | INCREMENT IDENTIFIER
               | DECREMENT IDENTIFIER
               | simpleExpression
               ;
    simpleExpression : simpleExpression LG_OR andExpression
                     | andExpression ;
    andExpression : andExpression LG_AND unaryRelExpression
                  | unaryRelExpression ;
    unaryRelExpression : NOT unaryRelExpression
                       | relExpression  ;
    relExpression : sumExpression GREATER_THAN sumExpression
                  | sumExpression LESSER_THAN sumExpression
                  | sumExpression LESS_EQ sumExpression
                  | sumExpression GREATER_EQ sumExpression
                  | sumExpression NOT_EQ sumExpression
                  | sumExpression EQUAL sumExpression
                  | sumExpression '|' sumExpression
```

```
                  | sumExpression '&' sumExpression
                  | sumExpression
                  ;
     sumExpression : sumExpression ADD term
                   | sumExpression SUBTRACT term
                   | term
                   ;
     term : term MULTIPLY unaryExpression
          | term DIVIDE unaryExpression
          | term MOD unaryExpression
          | unaryExpression
          ;
    unaryExpression : unaryOp %prec UMINUS unaryExpression
                    | factor ;

    unaryOp : UMINUS | '*'| UPLUS | '!' | '~' | '^' ;

    factor : IDENTIFIER | '(' expression ')' | const_type;
%%

void disp()
{
    printf("\n\tSymbol table");
    Display(SymbolTable);
    printf("\n\tConstant table");
    Display(ConstantTable);
}

#include "lex.yy.c"
int main(int argc , char *argv[]){

    SymbolTable = CreateTable();
    ConstantTable = CreateTable();

    // Open a file for parsing
    yyin = fopen(argv[1], "r");

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

    fclose(yyin);
    return 0;
}

int yyerror(char *msg)
{
```

```
  // Function to display error messages with line no and token
    printf("Line no: %d Error message: %s Token: %s\n", yylineno, msg, yytext);
    return 0;
}
```

The above code is the entire yacc script used for implementing the parser, it includes all the specifications for the parser, operator precedence and associativity. This script includes all the grammar rules and any relevant actions that have to be taken by the parser. The function used for displaying errors is also implemented here.

Table Initialization and display is also handled by this script by using the header file 'tables.h' which is the same as used in the lexical stage but with a change in the fields.

```
struct table_entry{

  int line_number;
  char *lexeme;
  double value;
  char* data_type;
  struct table_entry *next;
};
```

The above fields have been added to include data recognized by the parser at this stage.

Test Cases

Without Errors

Serial No	Test Case	Expected Output	Status
1.	`int main()` `{` `  int i,a =2,b =3;` `  for(i=0;i <= 2; i += 1)` `  {` `    a = a * 2;` `  }` `  while(i > 2)` `  {` `   b = b * 2;` `   i += 1;` `  }` `  do`	`Parsing complete.` `Symbol table` `----------------------------------------` `(lexeme, value, Data type, Line Number)` `( i, 2147483647.000000, INT, 3)` `( a, 2147483647.000000, INT, 3)` `( b, 2147483647.000000, INT, 3)` `----------------------------------------` `Constant table` `----------------------------------------` `(lexeme, value, Data type, Line Number)` `( 0, 0.000000, INT, 5)` `( 1, 1.000000, INT, 5)` `( 2, 2.000000, INT, 3)`	PASS

	{ a = a + 2; i = i + 1; } while(i != 2); }	(3, 3.000000, INT, 3) ----------	
2.	int main() { int i,a =2,b =3; if(a== 2) { if(b == 3) { a = a * 2; } else { if(a && b) { b = b * 2; } } } }	Parsing complete. Symbol table ---------- (lexeme, value, Data type, Line Number) (i, 2147483647.000000, INT, 3) (a, 2147483647.000000, INT, 3) (b, 2147483647.000000, INT, 3) ---------- Constant table ---------- (lexeme, value, Data type, Line Number) (2, 2.000000, INT, 3) (3, 3.000000, INT, 3) ----------	PASS
3.	int main() { int i,a =2,b =3; switch(a) { case 1: a = a*2; case 2: b = b - 2; break; default: a = a + 2; } return 0; }	Parsing complete. Symbol table ---------- (lexeme, value, Data type, Line Number) (i, 2147483647.000000, INT, 3) (a, 2147483647.000000, INT, 3) (b, 2147483647.000000, INT, 3) ---------- Constant table ---------- (lexeme, value, Data type, Line Number) (0, 0.000000, INT, 13) (1, 1.000000, INT, 7) (2, 2.000000, INT, 3) (3, 3.000000, INT, 3) ----------	PASS
4.	int main() { int i,a =2,b =3; int c, d= 8*7 && (a +	Parsing complete. Symbol table ----------	PASS

	b); char a = 's'; char *a = "Strings"; return 0; }	(lexeme, value, Data type, Line Number) (d, 2147483647.000000, INT, 5) (i, 2147483647.000000, INT, 3) (a, 2147483647.000000, CHAR, 3) (b, 2147483647.000000, INT, 3) (c, 2147483647.000000, INT, 5) ---------- Constant table ---------- (lexeme, value, Data type, Line Number) ("Strings", 0.000000, CHAR, 8) (0, 0.000000, INT, 10) (2, 2.000000, INT, 3) (3, 3.000000, INT, 3) (7, 7.000000, INT, 5) (8, 8.000000, INT, 5) ('s', 0.000000, CHAR, 7) ----------	
5.	int function(int a) { a = a * 2; return a; }	Parsing complete. Symbol table ---------- (lexeme, value, Data type, Line Number) (function, 2147483647.000000, INT, 1) (a, 2147483647.000000, INT, 1) ---------- Constant table ---------- (lexeme, value, Data type, Line Number) (2, 2.000000, INT, 3) ----------	PASS
6.	struct book { char *name; int price; } book1,book2,book3; union shelf { int loc; int size; }unit; int main() {	Parsing complete. Symbol table ---------- (lexeme, value, Data type, Line Number) (name, 2147483647.000000, CHAR, 3) (loc, 2147483647.000000, INT, 9) (book, 2147483647.000000, INT, 1) (shelf, 2147483647.000000, INT, 7) (price, 2147483647.000000, INT, 4) (size, 2147483647.000000, INT, 10) (unit, 2147483647.000000, INT, 11) (book1, 2147483647.000000, INT, 5) (book2, 2147483647.000000, INT, 5)	PASS

	int a; a = a * 2; }	(book3, 2147483647.000000, INT, 5) (a, 2147483647.000000, INT, 15) --- Constant table --- (lexeme, value, Data type, Line Number) (2, 2.000000, INT, 16) ---	
7.	int main() { int a,b = (a > 2)? 3: 4; }	Parsing complete. Symbol table --- (lexeme, value, Data type, Line Number) (a, 2147483647.000000, INT, 3) (b, 2147483647.000000, INT, 3) --- Constant table --- (lexeme, value, Data type, Line Number) (2, 2.000000, INT, 3) (3, 3.000000, INT, 3) (4, 4.000000, INT, 3) ---	PASS
8.	int main() { int a = 10; int x = 1,y = 0; for (i = 0; i < 10 ; i = i + 9){ for (g = 3 ; g < 10 ; g = g - 5) { printf("%d %d",i,g); } } diff = x - y; int rem = x % y; printf ("Total = %d \n", total); } int IncreaseBy10(int x) { return x + 10;	Parsing complete. Symbol table --- (lexeme, value, Data type, Line Number) (g, 2147483647.000000, INT, 7) (i, 2147483647.000000, INT, 6) (diff, 2147483647.000000, INT, 12) (x, 2147483647.000000, INT, 5) (y, 2147483647.000000, INT, 5) (rem, 2147483647.000000, INT, 13) (total, 2147483647.000000, INT, 14) (printf, 2147483647.000000, INT, 9) (IncreaseBy10, 2147483647.000000, INT, 17) (a, 2147483647.000000, INT, 4) --- Constant table --- (lexeme, value, Data type, Line Number)	PASS

	} 	(0, 0.000000, INT, 5) (1, 1.000000, INT, 5) (3, 3.000000, INT, 7) (5, 5.000000, INT, 7) (9, 9.000000, INT, 6) ("%d %d", 0.000000, CHAR, 9) ("Total = %d \n", 0.000000, CHAR, 14) (10, 10.000000, INT, 4) ----------	

With Errors

Serial No	Test Case	Expected Output	Status
1.	int main() { int i,a; for(i =0; i < 2; i = i + 1) { a = a * 2; //Missing bracket }	Line no: 10 Error message: syntax error Token: Parsing failed.	PASS
2.	int main() { int i,a; // missing semicolon a = a * 2 i = i + 1; }	Line no: 6 Error message: syntax error Token: i Parsing failed.	PASS
3.	int main() { int 9a; char s = 's'; }	Line no: 3 Error message: syntax error Token: ; Parsing failed.	PASS
4.	int main() { int i,a; // wrong assignment a + 2 = i; }	Line no: 5 Error message: syntax error Token: = Parsing failed.	PASS
5.	int main() { int i=3,a=2; if(a == 2) {	Line no: 12 Error message: syntax error Token: else Parsing failed.	PASS

	`` if(i == 1) a = a * 2; else a = a - 2; //extra invalid else else i = i + 2; } } ```		

Functionality and Implementation

The below section describes how the code identifies various constructs of the C language. All the rules are described using a context-free grammar. If the input sentence can be produced by using the grammar the sentence is accepted. Otherwise the parser displays a syntax error.

Preprocessor and Macro processor directives
Any C code has inclusion of header files in the beginning. In the current stage of parser the scanner does not take any action and does not return anything to the parser. But later during linking the compiler will link the header files. In case of any error in these directives the lexical analyser throws an exception and stops scanning.

1. Start Symbol

In our implementation of C grammar we have declared *program* as our start variable. This is done using the command *%start program*. In case the start symbol is not declared explicitly YACC automatically assumes first non terminal on the left side as the start symbol.

2. Declarations

A C program is made up of declarations. Primarily any code is made up of functions and global declaration. All functions can be derived using the rules mentioned in the section below.

3. Function Declarations

declarationList -> declarationList declaration | declaration *(Rule 1)*
Declaration -> varDeclaration | funDeclaration *(Rule 2)*

The above two rules can be used to produce more than one declarations which in turn can be functions or variable declarations. These variable declarations outside function declaration that is the ones derived independently from *declarationList* can be called global declarations.

4. Variable Declarations

varDeclaration -> typeSpecifier varDeclList ';' *(Rule 3)*
varDeclList -> varDeclList ',' varDeclInitialize
| varDeclInitialize ;
varDeclInitialize -> varDecId *(Rule 4*
| varDecId assignmentOperator assignmentExpression ;
*varDecId -> **IDENTIFIER*** *(Rule 5)*
*| **IDENTIFIER** '[' **INT_CONSTANT** ']' ;*

assignmentOperator -> = *(Rule 6)*
| **+=**
| **-=**
| ***=**
| **/+=**
| **%=** *;*

assignmentExpression -> conditionalStmt *(Rule 7)*
| *unaryExpression assignmentOperator assignmentExpression;*

The above rules are being used to construct variable declaration. Rule 3 helps us to declare multiple identifiers in a single statement and most importantly all the statements should end with a semicolon. The comma declaration is being realised with the rule 4. Many variables can be declared simultaneously and initialised as well.
Example - **int** a , b = 0;

The first rule produces **int** and *a , b = 0* is produced using second , third and fourth rule. For each variable declared , the declaration can either be only the **IDENTIFIER** or the **IDENTIFIER** can be initialised with a value. For variables involving initialisation rule 5 and 6 are used. The assignment can be done using any of the assignment operators as mentioned in rule 5. *assignmentExpression* is a non terminal which further covers all possibilities where the variable can be assigned using expressions and conditional statements. These cases are covered in following rules.

5. Function Declarations

funDeclaration -> typeSpecifier ***IDENTIFIER*** *'(' params ')' compoundStmt* *(Rule 8)*
| *typeSpecifier* **MAIN** *'(' params ')' compoundStmt*
| *noDefDeclare ;*
noDefDeclare -> typeSpecifier ***IDENTIFIER*** *'(' params ')' ';' ;* *(Rule 9)*
*funCall -> **IDENTIFIER** '(' args ')' statement;* *(Rule 10)*
args : argList | ;
argList : argList ',' expression | expression;
Params -> paramList | ; *(Rule 11)*
paramList -> paramList ',' paramTypeList | paramTypeList *(Rule 12)*
paramTypeList -> typeSpecifier paramId; *(Rule 13)*
paramId -> ***IDENTIFIER*** *|* ***IDENTIFIER*** *'[' ']'* *(Rule 14)*

The above rules are used to define as well as declare functions. Rule 8 is used for declarations. It produces all cases of valid declarations with return type , name and parameter list. A function call is a similar statement which does not include the return type and is produced by rule 9. To generate all valid cases of parameter list a non terminal *params* is used. To generate more than one parameter rule 9 and rule 10 work together. Each entry in the list is of the form type-specifier ***IDENTIFIER*** which is

generated by rule 11. Further to our grammar we have added the use of array as a parameter which is generated by rule 11 as type-specifier ***IDENTIFIER*** [ ].

6. Statements

statement -> expressionStmt | compoundStmt | selectionStmt (Rule 15)
| iterationStmt | jumpStmt | returnStmt | breakStmt | funCall ;
compoundStmt -> '{' localDeclarations statementList '}' ; (Rule 16)
localDeclarations -> localDeclarations varDeclaration | ; (Rule 17)
statementList -> statementList statement (Rule 18)
| ;

The above rule is used to generate any kind of statements in C. These statements usually arise inside functions. The function generates curly brackets using *compoundStmt* as seen in rule 7.
The *compoumdStmt* on its own can produce a list of statements denoted by *statementList.* This function can have local declarations which are produced by *localDeclarations.* This non terminal in turn produces many variable declaration using rule 3. The staementList is made up of multiple statements or can also be empty. Empty statement is derived from the second part of rule 16.

7. If-else Statements

selectionStmt -> IF '(' simpleExpression ')' statement %prec IFX (Rule 19)
| IF '(' simpleExpression ')' statement ELSE statement;
| SWITCH '(' simpleExpression ')' statement;

This rule is used to verify the syntax of all if-else statements. This rule also handles the dangling else problem, this is done using %nonassoc declarations in the definition section for IFX and ELSE, and the higher precedence to IFX which solves the problem of matching the else statement.
simpleExpression is a non terminal which is further used to derive all possible inputs to if statement for evaluation.
statement is another non terminal which is used to signify all possible blocks of code which can come after if statement and after else statement.

8. Iterative Statements

*iterationStmt -> **WHILE** '(' simpleExpression ')' statement (Rule 20)*
*| **DO** statement WHILE '(' expression ')' ';'*
*| **FOR** '(' optExpression ';' optExpression ';' optExpression ')'*
statement;

The *iterativeStmt* production rule is used for identifying all iterative programs in the C language, here we have included rules for While loops, for loops and do while loops.

The rules follow the simple syntactical specifications of C. The ‘statement’ non terminal is used for all code blocks that will follow the loop statements. The *expression* and *simpleExpression* non terminals are used for identifying statements inside the loop. The *optExpression* non terminal is used in for loops to allow for empty statements, as a for loop in C need not have either of the 3 statements for it to be syntactically legal.

9. Expressions

expressionStmt -> expression ';' | ';' ; *(Rule 21)*
*expression -> **IDENTIFIER** assignmentOperator expression* *(Rule 22)*
*| INCREMENT **IDENTIFIER***
*| DECREMENT **IDENTIFIER***
| simpleExpression
;

simpleExpression -> simpleExpression LG_OR andExpression ‘ *(Rule 23)*
| andExpression ;

andExpression -> andExpression LG_AND unaryRelExpression *(Rule 24)*
| unaryRelExpression ;

unaryRelExpression -> NOT unaryRelExpression *(Rule 25)*
| relExpression ;

relExpression -> sumExpression >= sumExpression *(Rule 26)*
| sumExpression <= sumExpression
| sumExpression < sumExpression
| sumExpression > sumExpression
| sumExpression != sumExpression
| sumExpression == sumExpression
| sumExpression
;

sumExpression -> sumExpression + term *(Rule 27)*
| sumExpression -term
| term
;

*term -> term * unaryExpression* *(Rule 28)*
| term / unaryExpression
| term % unaryExpression
| unaryExpression
;

unaryExpression ->unaryOp %prec UMINUS unaryExpression *(Rule 29)*
| factor ;

unaryOp -> UMINUS | ''| UPLUS | '!' | '~' | '^' | '&' ;* *(Rule 30)*
*factor -> **IDENTIFIER** | '(' expression ')' | const_type;* *(Rule 31)*

The above rules from rule number 19 to 30 are used to derive a large subset of C allowed expressions. The second part of rule 20 is used to parse a single semicolon in a statement as permitted by C. All other expressions proceed through the first production rule. Expression can be an assignment expression or a simple expression. The above grammar thus allows multi-assignment statements. Simple expressions are produced using rule 22 and further. It expresses all expressions as logical **OR** of **AND** statements. All AND statements further made up of *unaryExpressions*. *unaryExpressions* can be made of relational expressions denoted by *relExpressions*. All the relational operators **< , > , <= , >= , != , ==** are covered in this section. *relExpressions* have *sumExpression* as their operands. *sumExpression* are further written as sum and subtraction of terms. Terms are expressions which multiplication , division and modulus as terminals. One of the non terminals on the rhs for rule 27 is *unaryExpression*. Unary expressions can be unary operators and one operand. Thus the rule 28. All the unary operators implemented in our parser on being produced in rule 28. In the end , the nesting to any level in the expressions with the second part in rule 30. Further , the rule 30 produces terminals **IDENTIFIER** and const_type.

10. Structure and Union

strucUniDecl -> STRUCT_UNI IDENTIFIER '{' declarationRecur '}' idBlock ';' ;
(Rule 33)

idBlock -> idBlock ',' IDENTIFIER *(Rule 34)*
| idBlock ',' IDENTIFIER '[' INT_CONSTANT ']'
| IDENTIFIER
| IDENTIFIER '[' INT_CONSTANT ']';

declarationRecur -> declarationRecur declaration | declaration ; *(Rule 35)*

The above production rules are used to identify structure declarations and union declarations, as both these data structures follow the same rules a single token has been used for both of them **STRUCT_UNI**.

11. Switch Statement

*selectionStmt -> **IF** '(' simpleExpression ')' statement %prec IFX* *(Rule 19)*
*| **IF** '(' simpleExpression ')' statement **ELSE** statement*
*| **SWITCH** '(' simpleExpression ')' statement;*
statement -> labeledStmt | expressionStmt | compoundStmt *(Rule 15)*
| selectionStmt | iterationStmt | jumpStmt | returnStmt
| breakStmt | funCall ;

*labeledStmt -> **IDENTIFIER** ':' statement* *(Rule 36)*

| *CASE conditionalStmt ':' statement* | *DEFAULT ':' statement*

The above three rules are used to derive all constructs of switch statement. The syntax of switch statement in C is as follows -

```
switch(x){
        case a: { break; }
        default: { break; }
}
```

To identify a switch statement when the scanner returns **SWITCH** token the control shifts from *selectionStmt* to statement. Statement then selects the rule corresponding to *labeledStmt* which further matches any number of CASE statements with their respective code blocks. These code blocks for each case are derived by the non terminal statement in rule 36.

12. Conditional Statement

conditionalStmt -> simpleExpression '?' expression ':' conditionalStmt (Rule 37)
| *simpleExpression;*

The parser implementation supports statements of ternary operators or conditional statements. The start state of above rule can be reached either during variable assignment (rule 5 and rule 7) . After the start state is reached the syntax of the statement is verified using rule 37. The rule also allows nested conditional statement as we can see in the above rule.

Results and Future Work

Results

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 functions present in the C program are also identified as functions and not as identifiers. The parser generates error messages in case of any syntactical errors in the test program.

Input 1: Sample Programs with most features of C covered

```
int main()
{
    int a,b;
    int *c;
    int d;
    int a, b= 2*2 - 8 && 9;
    d = ++d;
    a=a-b;

    while(a > 0)
    {
        a = a - 1;
    }

    if(a == 2)
    {
        if( b == 2 + a)
        {
            a = a - 2;
        }
        else
        {
            a = a + 2;
        }

    }
}

int function(){
    // Here's a comment
    char *a= "abcd";
    int p[10];

    return 0;
}
```

Output 1: Parser Output

```
Parsing complete

        Symbol table

----------------------------------------

        (lexeme, value, Data type, Line Number)
        (    d, 2147483647.000000, INT, 5)
        (    p, 2147483647.000000, INT, 32)
        (function, 2147483647.000000, INT, 29)
        (    a, 2147483647.000000, CHAR, 3)
        (    b, 2147483647.000000, INT, 3)
        (    c, 2147483647.000000, INT, 4)
----------------------------------------

        Constant table

----------------------------------------

        (lexeme, value, Data type, Line Number)
        (    0, 0.000000, INT, 10)
        (    1, 1.000000, INT, 12)
        (    2, 2.000000, INT, 6)
        (    8, 8.000000, INT, 6)
        (    9, 9.000000, INT, 6)
        ("abcd", 0.000000, CHAR, 31)
        (   10, 10.000000, INT, 32)
----------------------------------------
```

As seen in the above test case the parsing has completed for the general C program that consists of loop constructs, strings, arrays, conditional statements, functions,different operators such as assignment, arithmetic, relational, etc. The dangling else issue has also been accounted for by matching the else with the most recent unmatched if. Once the parsing has been completed the identifiers and constants are identified and added to the symbol and constant table respectively as seen above with their names, data types and the line numbers they are defined in.

Input 2: Sample C program missing closing bracket

```
int main()
{
    int b;
    int a = b + 5;

    //Missing closing bracket
   for(i =0 ; i < 5; i = i + 1
   {
       a= a *2;
   }

}

// Missing Parenthesis for Function
int func()
{
    int a  =  2 * 2;
```

Output 2: Parser Output

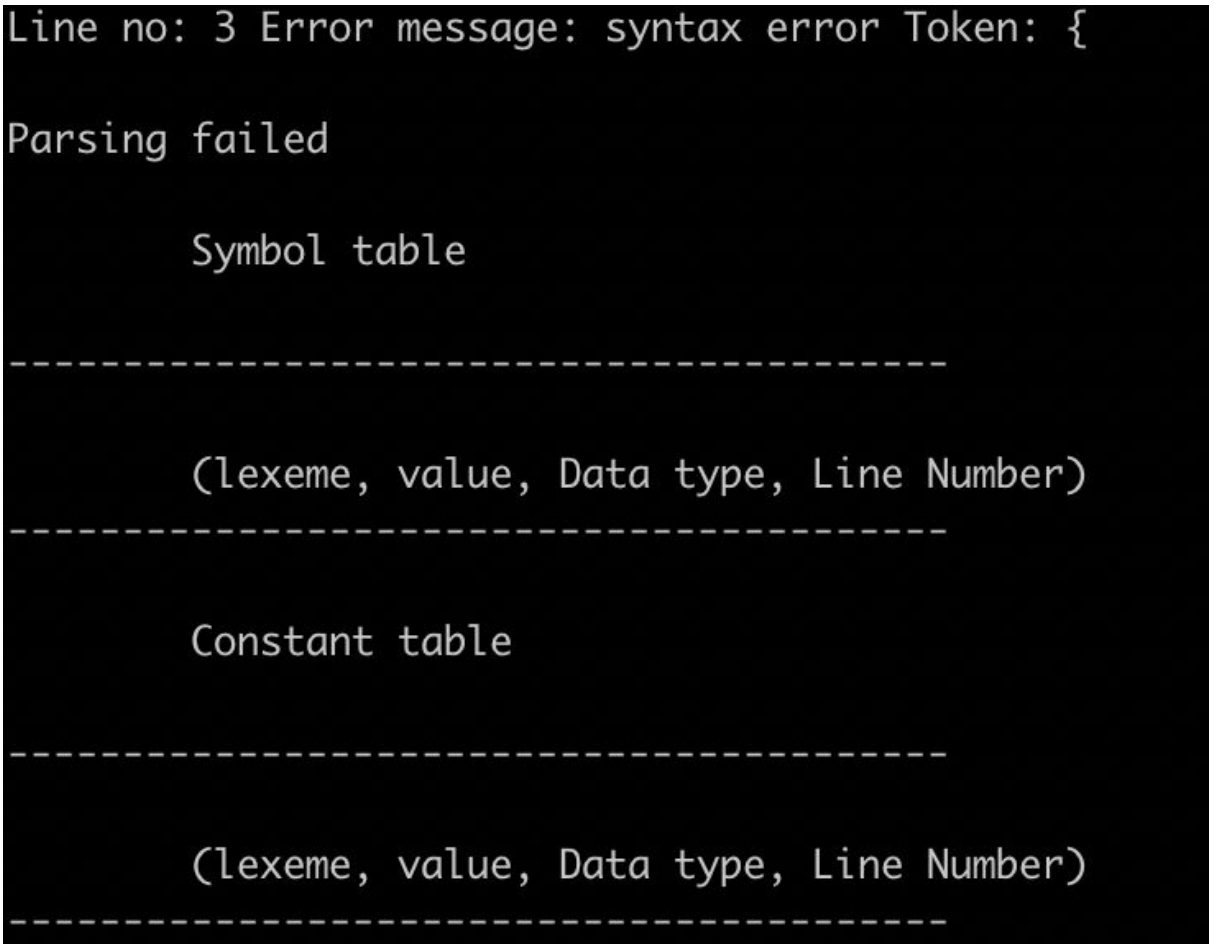
```
Line no: 9 Error message: syntax error Token: {

Parsing failed
```

As seen above the parsing failed due to an error created due to an invalid syntax as the for loop hasn't been closed with a closing ')' bracket. Before the missing parenthesis can be identified our parser throws an error due to the missing ')' . The error is indicated with its line numbers.

Input 3: Missing semicolon

```
#include <stdio.h>

struct{
    int a;
    int b;
}abc;

int main()
{
    //Missing semicolon
    int a

    a = a * 2;

}
```

Output 3: Parser Output



As indicated above the parsing for the above test case failed due to a missing semicolon. And this has been indicated by the parser with its line number.

Input 4: Sample C program with nested code blocks

```
int main()
{

  int a = 10;
  int x = 1,y = 0;
  for ( i = 0; i < 10 ; i = i + 9){
    for ( g = 3 ; g < 10 ; g = g - 5)
    {
      printf("%d %d",i,g);
    }
  }
  diff = x - y;
  int rem = x % y;
  printf ("Total = %d \n", total);

}
```

Output 4: Parser Output

```
Parsing complete.

        Symbol table

---------------------------------------

        (lexeme, value, Data type, Line Number)
        (    g, 2147483647.000000, INT, 7)
        (    i, 2147483647.000000, INT, 6)
        ( diff, 2147483647.000000, INT, 12)
        (    x, 2147483647.000000, INT, 5)
        (    y, 2147483647.000000, INT, 5)
        (  rem, 2147483647.000000, INT, 13)
        (total, 2147483647.000000, INT, 14)
        (printf, 2147483647.000000, INT, 9)
        (    a, 2147483647.000000, INT, 4)
---------------------------------------

        Constant table

---------------------------------------

        (lexeme, value, Data type, Line Number)
        (    0, 0.000000, INT, 5)
        (    1, 1.000000, INT, 5)
        (    3, 3.000000, INT, 7)
        (    5, 5.000000, INT, 7)
        (    9, 9.000000, INT, 6)
        ("%d %d", 0.000000, CHAR, 9)
        ("Total = %d \n", 0.000000, CHAR, 14)
        (   10, 10.000000, INT, 4)
---------------------------------------
```

Future work

The yacc script we have used for the parser implements a subset of the C language, further work can be done to include more features, we had to implement a small subset because of time constraints.

More work can be done on lex script and yacc script to work with value assignment. More grammar can be added for including more functionality (bitwise operators is one example).

References

1. https://www.cs.dartmouth.edu/~mckeeman/cs48/mxcom/doc/Ccfg.html
2. http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf
3. http://dinosaur.compilertools.net/yacc/