Parser for C language

Submitted as part of course requirement for Compiler Design, CS304

Submitted by:
Feyaz Baker 181CO119
Nuthan Kumar 181CO137
Nihar Rai 181CO235
Arunava Mukhoti 181CO210

Abstract

This report contains the details of the tasks finished as a part of Phase Two of the Compiler Design Lab. This phase involved the development of a Parser for C language which makes use of the C lexer to parse the given C input file. The previous work was concentrated at developing a lexical analyzer using a flex script to generate a stream of tokens and populate the symbol table. The lexical analyzer is only able to handle errors involving invalid tokens. A parser is needed to handle various other errors that might exist like syntax errors. After the lexical phase, the compiler enters the syntax analysis phase where a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. This is done by a parser. 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.

Index

Abstract **2**

Index **3**

List of figures and tables **4**

Figures 4

Tables 4

Introduction **5**

Parser 5

Yacc script 5

Code review **7**

Scanner code (flex script) 7

Scanner code explanation 16

Declaration section 16

Rules section 17

Parser generator (Yacc script) 17

Parser generator explanation 27

Declaration section 27

Rules section 28

Auxiliary routines section 28

Test cases **29**

Error free code 29

Test 1 29

Test 2 30

Test 3 31

Code with errors 32

Test 1 32

Test 2 33

Test 3 33

Implementation **34**

First table for nonterminals **35**

Follow table for nonterminals **37**

Limitations and future work **42**

References **42**

List of figures and tables

Figures

1. Figure 1: Output for Error free code, Test 1
2. Figure 2: Output for Error free code, Test 2
3. Figure 3: Output for Error free code, Test 3
4. Figure 4: Output for Code with errors, Test 1
5. Figure 5: Output for Code with errors, Test 2
6. Figure 6: Output for Code with errors, Test 3

Tables

1. Table 1: First table for non-terminals
2. Table 2: Follow table for non-terminals

Introduction

Parser

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 input to the parser is the stream of tokens generated by the lexical analyser in the form of a symbol table which was populated in the lexical analysis phase. The parser then identifies any syntax errors present. The parser has to parse the entire code even if it finds errors. Hence it should be able to recover from commonly occurring errors while reporting it.
The type of errors identified can be broadly classified as:

- Structural errors
- Missing identifiers
- Wrong keywords
- Unbalanced parentheses

Parser converts input tokens into parse trees with the help of the grammar defined to identify these tokens. There are two types of parsers :

1. Top-down parser : Top-down Parser is the parser which generates a parse tree for the given input string with the help of grammar productions by expanding the non-terminals i.e., it starts from the start symbol and ends on the terminals.
2. Bottom-up parser : Bottom-up Parser is the parser that generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e., it starts from non-terminals and ends on the start symbol. It uses the reverse of the rightmost derivation.

Yacc script

Yacc (yet another compiler compiler) is a grammar parser and parser generator. That is, it is a program that reads a grammar specification and generates code that is able to organize input tokens in a syntactic tree in accordance with the grammar. Yacc provides a comprehensive tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input.

A parser calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then the user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions. The lexer can be used to make a simple parser. But it needs making extensive use of the user-defined states. The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the wrong data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the wrong data.

The structure of a yacc script is the following :

```
declarations section

%%

rules section

%%

Auxiliary routines
```

The definition part includes information about the tokens used in the syntax definition. It can include C code external to the definition of the parser and variable declarations, within **%{** and **%}** in the first column.

The rules part contains grammar definition. Actions are C code in { } and can be embedded inside.

The auxiliary section contains only C code. It contains function definitions of every function used in the rules section. It also has a main function that calls the yyparse() method.

Code review

Scanner code (flex script)

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

   int nesting = 0;
   #define TERM_RED    "\x1b[31m"
   #define TERM_GRN    "\x1b[32m"
   #define TERM_YLW    "\x1b[33m"
   #define TERM_BLU    "\x1b[34m"
   #define TERM_MGT    "\x1b[35m"
   #define TERM_CYN    "\x1b[36m"
   #define TERM_DEF    "\x1b[0m"
   #define tablesize   1007

   struct symboltable
   {
       char name[20];
       char class[20];
       char type[20];
       char value[20];
       int lineno;
       int nesting;
       int length;
   }ST[tablesize];

   struct constanttable
   {
       char name[20];
       char type[20];
       int length;
       int nesting;
   }CT[tablesize];

   int hash(char *str)
   {
       int value = 0;
       for(int i = 0 ; i < strlen(str) ; i++)
       {
           value = 10*value + (str[i] - 'A');
```

```
        value = value % tablesize;
        while(value < 0)
            value = value + tablesize;
    }
    return value;
}

int lookupST(char *str)
{
    int value = hash(str);
    if(ST[value].length == 0)
    {
        return 0;
    }
    else if(strcmp(ST[value].name,str)==0)
    {
        return 1;
    }
    else
    {
        for(int i = value + 1 ; i!=value ; i = (i+1)%tablesize)
        {
            if(strcmp(ST[i].name,str)==0)
            {
                return 1;
            }
        }
        return 0;
    }
}

int lookupCT(char *str)
{
    int value = hash(str);
    if(CT[value].length == 0)
        return 0;
    else if(strcmp(CT[value].name,str)==0)
        return 1;
    else
    {
        for(int i = value + 1 ; i!=value ; i = (i+1)%tablesize)
        {
            if(strcmp(CT[i].name,str)==0)
            {
                return 1;
            }
        }
        return 0;
    }
}

void insertST(char *str1, char *str2)
{
    if(lookupST(str1))
    {
        return;
    }
    else
    {
        int value = hash(str1);
        if(ST[value].length == 0)
        {
            strcpy(ST[value].name,str1);
            strcpy(ST[value].class,str2);
            ST[value].length = strlen(str1);
            ST[value].nesting=nesting;
            insertSTline(str1,yylineno);
            return;
        }

        int pos = 0;

        for (int i = value + 1 ; i!=value ; i = (i+1)%tablesize)
        {
            if(ST[i].length == 0)
            {
                pos = i;
                break;
            }
        }

        strcpy(ST[pos].name,str1);
        strcpy(ST[pos].class,str2);
        ST[pos].length = strlen(str1);
    }
}

void insertSTtype(char *str1, char *str2)
{
    for(int i = 0 ; i < tablesize ; i++)
    {
        if(strcmp(ST[i].name,str1)==0)
        {
            strcpy(ST[i].type,str2);
        }
    }
```

```
}

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

void insertSTline(char *str1, int line)
{
    for(int i = 0 ; i < tablesize ; i++)
    {
        if(strcmp(ST[i].name,str1)==0)
        {
            ST[i].lineno = line;
        }
    }
}

void insertCT(char *str1, char *str2)
{
    if(lookupCT(str1))
        return;
    else
    {
        int value = hash(str1);
        if(CT[value].length == 0)
        {
            strcpy(CT[value].name,str1);
            strcpy(CT[value].type,str2);
            CT[value].length = strlen(str1);
            CT[value].nesting = nesting;
            return;
        }

        int pos = 0;

        for (int i = value + 1 ; i!=value ; i = (i+1)%tablesize)
        {
            if(CT[i].length == 0)
            {
                pos = i;
                break;
                }
            }

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

    void printST()
    {
        printf("%10s | %15s | %10s | %15s | %10s | %10s\n","SYMBOL", "CLASS",
"TYPE","VALUE", "LINE NO", "NESTING");
        for(int i=0;i<81;i++) {
            printf("-");
        }
        printf("\n");
        for(int i = 0 ; i < tablesize ; i++)
        {
            if(ST[i].length == 0)
            {
                continue;
            }
            printf("%10s | %15s | %10s | %15s | %10d | %10d\n",ST[i].name,
ST[i].class, ST[i].type, ST[i].value, ST[i].lineno, ST[i].nesting);
        }
    }


    void printCT()
    {
        printf("%15s | %15s | %10s\n","NAME", "TYPE", "NESTING");
        for(int i=0;i<81;i++) {
            printf("-");
        }
        printf("\n");
        for(int i = 0 ; i < tablesize ; i++)
        {
            if(CT[i].length == 0)
                continue;

            printf("%15s | %15s  | %10d\n",CT[i].name, CT[i].type,
CT[i].nesting);
        }
    }
    char curid[20];
    char curtype[20];
    char curval[20];
```

```
%}

DE "define"
IN "include"

%%
\n  {yylineno++;}
([#][" "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|"
"|"\t"]  { }
([#][" "]*({DE})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"]
{ }
\/\/(.*)
{ }
\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/
{ }
[ \n\t] ;
";"             { return(';'); }
","             { return(','); }
("{")           { nesting++;    return('{'); }
("}")           { nesting--;    return('}'); }
"("             { return('('); }
")"             { return(')'); }
("["|"<:")      { return('['); }
("]"|":>")      { return(']'); }
":"             { return(':'); }
"."             { return('.'); }

"auto"          { strcpy(curtype,yytext); insertST(yytext, "keyword");
return AUTO; }
"extern"        { strcpy(curtype,yytext); insertST(yytext, "keyword");
return EXTERN; }
"register"      { strcpy(curtype,yytext); insertST(yytext, "keyword");
return REGISTER; }
"static"        { strcpy(curtype,yytext); insertST(yytext, "keyword");
return STATIC; }

"int"           { strcpy(curtype,yytext); insertST(yytext, "keyword");
return INT;}
"char"          { strcpy(curtype,yytext); insertST(yytext, "keyword");
return CHAR;}
"float"         { strcpy(curtype,yytext); insertST(yytext, "keyword");
return FLOAT;}
"double"        { strcpy(curtype,yytext); insertST(yytext, "keyword");
return DOUBLE;}
"void"          { strcpy(curtype,yytext); insertST(yytext, "keyword");
return VOID;}
```

```
"long"          { strcpy(curtype,yytext); insertST(yytext, "keyword");
return LONG;}
"short"         { strcpy(curtype,yytext); insertST(yytext, "keyword");
return SHORT;}
"signed"        { strcpy(curtype,yytext); insertST(yytext, "keyword");
return SIGNED;}
"unsigned"      { insertST(yytext, "keyword");  return UNSIGNED;}

"if"            { insertST(yytext, "keyword"); return IF;}
"else"          { insertSTline(yytext, yylineno); insertST(yytext,
"keyword"); return ELSE;}

"while"         { insertST(yytext, "keyword"); return WHILE;}
"do"            { insertST(yytext, "keyword"); return DO;}
"for"           { insertST(yytext, "keyword"); return FOR;}

"break"         { insertST(yytext, "keyword"); return BREAK;}
"continue"      { insertST(yytext, "keyword"); return CONTINUE;}
"return"        { insertST(yytext, "keyword");  return RETURN;}

"sizeof"        { insertST(yytext, "keyword");  return SIZEOF;}
"struct"        { strcpy(curtype,yytext); insertST(yytext, "keyword");
return STRUCT;}


"++"            { return increment_operator; }
"--"            { return decrement_operator; }
"<<"            { return leftshift_operator; }
">>"            { return rightshift_operator; }
"<="            { return leq_operator; }
"<"             { return le_operator; }
">="            { return geq_operator; }
">"             { return ge_operator; }
"=="            { return equality_operator; }
"!="            { return inequality_operator; }
"&&"            { return AND_operator; }
"||"            { return OR_operator; }
"^"             { return caret_operator; }
"*="            { return multiplication_assignment_operator; }
"/="            { return division_assignment_operator; }
"%="            { return modulo_assignment_operator; }
"+="            { return addition_assignment_operator; }
"-="            { return subtraction_assignment_operator; }
"&"             { return amp_operator; }
"!"             { return exclamation_operator; }
"~"             { return tilde_operator; }
"-"             { return subtract_operator; }
```

```
"+"             { return add_operator; }
"*"             { return multiplication_operator; }
"/"             { return division_operator; }
"%"             { return modulo_operator; }
"|"             { return pipe_operator; }
"="             { return assignment_operator;}

\"[^\n]*\"/[;|,|\)]             {strcpy(curval,yytext);
insertCT(yytext,"string constant"); return string_constant;}
\'[A-Z|a-z]\'/[;|,|\)|:]        {strcpy(curval,yytext);
insertCT(yytext,"character constant"); return character_constant;}
[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[  {strcpy(curid,yytext); insertST(yytext,
"array Identifier");  return identifier;}
[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]
{strcpy(curval,yytext); insertCT(yytext, "number constant"); return
integer_constant;}
([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^]
{strcpy(curval,yytext); insertCT(yytext, "floating constant"); return
float_constant;}
[A-Za-z_][A-Za-z_0-9]* {strcpy(curid,yytext);insertST(yytext,"identifier");
return identifier;}

(.?) {
       if(yytext[0]=='#')
       {
           printf("Error in Pre-Processor directive at line no.
%d\n",yylineno);
       }
       else if(yytext[0]=='/')
       {
           printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);
       }
       else if(yytext[0]=='"')
       {
           printf("ERR_INCOMPLETE_STRING at line no. %d\n",yylineno);
       }
       else
       {
           printf("ERROR at line no. %d\n",yylineno);
       }
       printf("%s\n", yytext);
       return 0;
}

%%
```

Scanner code explanation

Scanner is a flex script that detects tokens from the input source code and returns it to the parser. It also populates the symbol table and constant table.

Declaration section

In this section, we have given all the header files required, function declarations and definitions and data structures required for the scanner.
The data structures used are :

1. symboltable : This is the structure that defines characteristics of each entry of the symbol table. ST[] is an array of this structure which is used to store the entries of the symbol table.
2. constanttable : This is the structure that defines characteristics of each entry of the constant table. CT[] is an array of this structure which is used to store the entries of the constant table.
3. curid : An array of characters used to store the identifier name which was just scanned.
4. curtype : An array of characters used to store the data type which was just scanned.
5. curval : An array of characters used to store the constant value which was just scanned.

Functions used are :

1. hash(char *str) : A hash function that takes a string as a parameter and generates a key.
2. lookupST(char *str) : A function that takes an identifier name as a parameter and checks if an entry corresponding to that identifier already exists in the symbol table. Returns 0 if it doesn't exist and 1 if it already exists.
3. lookupCT(char *str) : A function that takes a constant as a parameter and checks if an entry corresponding to that constant already exists in the constant table. Returns 0 if it doesn't exist and 1 if it already exists.
4. insertST(char *str1, char *str2) : creates an entry in the symbol table if an entry doesn’t exist for that identifier.
5. insertSTtype(char *str1, char *str2) : enters the data type of the identifier in the symbol table.
6. insertSTvalue(char *str1, char *str2) : enters the value of the identifier in the symbol table.
7. insertSTline(char *str1, char *str2) : enters the line number of the identifier in the symbol table.
8. insertCT(char *str1, char *str2) : creates an entry in the constant table if an entry doesn’t exist for that constant.
9. printST() : prints the contents of the symbol table.

10. printCT() : prints the contents of the constant table.

Rules section

This section contains the regular expressions to identify tokens. The actions to be taken when a token is encountered is also specified adjacent to the regular expression.

Parser generator (Yacc script)

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

   #define TERM_RED    "\x1b[31m"
   #define TERM_GRN    "\x1b[32m"
   #define TERM_YLW    "\x1b[33m"
   #define TERM_BLU    "\x1b[34m"
   #define TERM_MGT    "\x1b[35m"
   #define TERM_CYN    "\x1b[36m"
   #define TERM_DEF    "\x1b[0m"

   extern char curid[20];
   extern char curtype[20];
   extern char curval[20];

%}

%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
%token RETURN MAIN
%token AUTO
%token REGISTER
%token EXTERN
%token STATIC
%token VOID
%token WHILE FOR DO
%token BREAK CONTINUE
%token ENDIF
%expect 2
```

```
%token identifier
%token integer_constant string_constant float_constant character_constant

%nonassoc IF
%nonassoc ELSE

%right modulo_assignment_operator
%right multiplication_assignment_operator division_assignment_operator
%right addition_assignment_operator subtraction_assignment_operator
%right assignment_operator

%left OR_operator
%left AND_operator
%left pipe_operator
%left caret_operator
%left amp_operator
%left equality_operator inequality_operator
%left leq_operator le_operator geq_operator ge_operator
%left leftshift_operator rightshift_operator
%left add_operator subtract_operator
%left multiplication_operator division_operator modulo_operator

%right SIZEOF
%right tilde_operator exclamation_operator
%left increment_operator decrement_operator


%start code

%%
code
           : decl_list;

decl_list
           : decl D

D
           : decl_list
           | ;

decl
           : var_decl
           | funct_decl
           | struct_decl;

var_decl
           : type_specifier var_decl_list ';'
           | structure_decl;
```

```
var_decl_list
           : var_decl_identifier V;

V
           : ',' var_decl_list
           | ;

var_decl_identifier
           : identifier { ins(); } var_decl_identification;

var_decl_identification
           : identifier_array_type
           | assignment_operator expression ;

identifier_array_type
           : '[' initialization_params
           | ;

initialization_params
           : integer_constant ']' initialization
           | ']' string_initialization;

initialization
           : string_initialization
           | array_initialization
           | ;

storage_class_specifier
           : AUTO
           | STATIC
           | EXTERN
           | REGISTER
           | ;

type_specifier
           : storage_class_specifier type_spec
           | VOID ;

type_spec
           : INT
           | CHAR
           | FLOAT
           | DOUBLE
           | LONG long_grammar
           | SHORT short_grammar
           | UNSIGNED unsigned_grammar
           | SIGNED signed_grammar
           ;


unsigned_grammar
           : INT
           | LONG long_grammar
           | SHORT short_grammar
           | ;

signed_grammar
           : INT | LONG long_grammar | SHORT short_grammar | ;

long_grammar
           : INT | ;

short_grammar
           : INT | ;

struct_decl
           : STRUCT identifier { ins(); } '{' V1  '}' ';';

V1 : var_decl V1 | ;

structure_decl
           : STRUCT identifier var_decl_list;


funct_decl
           : funct_decl_type funct_decl_params;

funct_decl_type
           : type_specifier identifier '('  { ins();};

funct_decl_params
           : params ')' stmt;

params
           : parameters_list | ;

parameters_list
           : type_specifier parameters_identifier_list;

parameters_identifier_list
           : param_identifier parameters_identifier_list_parts;

parameters_identifier_list_parts
           : ',' parameters_list
           | ;
```

```
param_identifier
           : identifier { ins(); } param_identifier_parts;

param_identifier_parts
           : '[' ']'
           | ;

stmt
           : expression_stmt
           | compound_stmt
           | conditional_stmt
           | iterative_stmt
           | flow_stmt
           | var_decl;

compound_stmt
           : '{' statement_list '}' ;

statement_list
           : stmt statement_list
           | ;

expression_stmt
           : expression ';'
           | ';' ;

conditional_stmt
           : IF '(' simple_expression ')' stmt conditional_stmt_parts;

conditional_stmt_parts
           : ELSE stmt
           | ;

iterative_stmt
           : WHILE '(' simple_expression ')' stmt
           | FOR '(' expression ';' simple_expression ';' expression ')'
           | DO stmt WHILE '(' simple_expression ')' ';';

flow_stmt
           : CONTINUE ';'
           | BREAK ';'
           | RETURN ';'
           | RETURN expression ';'

string_initialization
           : assignment_operator string_constant { insV(); };
```

```
array_initialization
           : assignment_operator '{' array_int_decl '}';

array_int_decl
           : integer_constant array_int_decl_parts;

array_int_decl_parts
           : ',' array_int_decl
           | ;

expression
           : mutable expression_parts
           | simple_expression ;

expression_parts
           : assignment_operator expression
           | addition_assignment_operator expression
           | subtraction_assignment_operator expression
           | multiplication_assignment_operator expression
           | division_assignment_operator expression
           | modulo_assignment_operator expression
           | increment_operator
           | decrement_operator ;

simple_expression
           : and_expression simple_expression_parts;

simple_expression_parts
           : OR_operator and_expression simple_expression_parts | ;

and_expression
           : unary_relation_expression and_expression_parts;

and_expression_parts
           : AND_operator unary_relation_expression and_expression_parts
           | ;

unary_relation_expression
           : exclamation_operator unary_relation_expression
           | regular_expression ;

regular_expression
           : sum_expression regular_expression_parts;

regular_expression_parts
           : relational_operators sum_expression
           | ;
```

```
relational_operators
           : geq_operator
           | leq_operator
           | ge_operator
           | le_operator
           | equality_operator
           | inequality_operator ;

sum_expression
           : sum_expression sum_operators term
           | term ;

sum_operators
           : add_operator
           | subtract_operator ;

term        : factor Ax;

Ax          : MULOP factor Ax
           | ;

MULOP
           : multiplication_operator
           | division_operator
           | modulo_operator ;

factor
           : immutable
           | mutable ;


mutable     : identifier Bx;

Bx          : mutable_parts Bx
           | ;

mutable_parts
           : '[' expression ']'
           | '.' identifier;

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

call
           : identifier '(' arguments ')';

arguments
```

```
           : arguments_list | ;

arguments_list
           : expression E;

E
           : ',' expression E
           | ;

constant
           : integer_constant      { insV(); }
           | string_constant       { insV(); }
           | float_constant        { insV(); }
           | character_constant    { insV(); };

%%

extern FILE *yyin;
extern int yylineno;
extern int nesting;
extern char *yytext;
void insertSTtype(char *,char *);
void insertSTvalue(char *, char *);
void incertCT(char *, char *);
void printST();
void printCT();

int main(int argc , char **argv)
{
   yyin = fopen(argv[1], "r");
   yyparse();

   if(flag == 0)
   {
       printf(TERM_GRN "Parsing completed succesfully" TERM_DEF "\n");
       printf(TERM_CYN "SYMBOL TABLE" TERM_DEF "\n", " ");

       printST();

       printf("\n\n" TERM_CYN "CONSTANT TABLE" TERM_DEF "\n");
       printCT();
   }
}

void yyerror(char *s)
{
   printf("%d %s %s\n", yylineno, s, yytext);
   flag=1;
   printf(TERM_RED "Parsing Failed: Invalid input\n" TERM_DEF);
}

void ins()
{
   insertSTtype(curid,curtype);
}

void insV()
{
   insertSTvalue(curid,curval);
}

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

Parser generator explanation

Declaration section

In this section we have included all the necessary header files, function declarations and flags that were needed in the code. Between the declaration and rules section we have listed all the tokens which are returned by the lexer according to the precedence order. We also declared the operators here according to their associativity and precedence.

Rules section

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. The grammar productions does the syntactical analysis of the code.

Auxiliary routines section

In this section the parser links the extern functions, variables declared in the lexer and external files generated by the lexer. The main function takes the input source code file and prints the final symbol table.

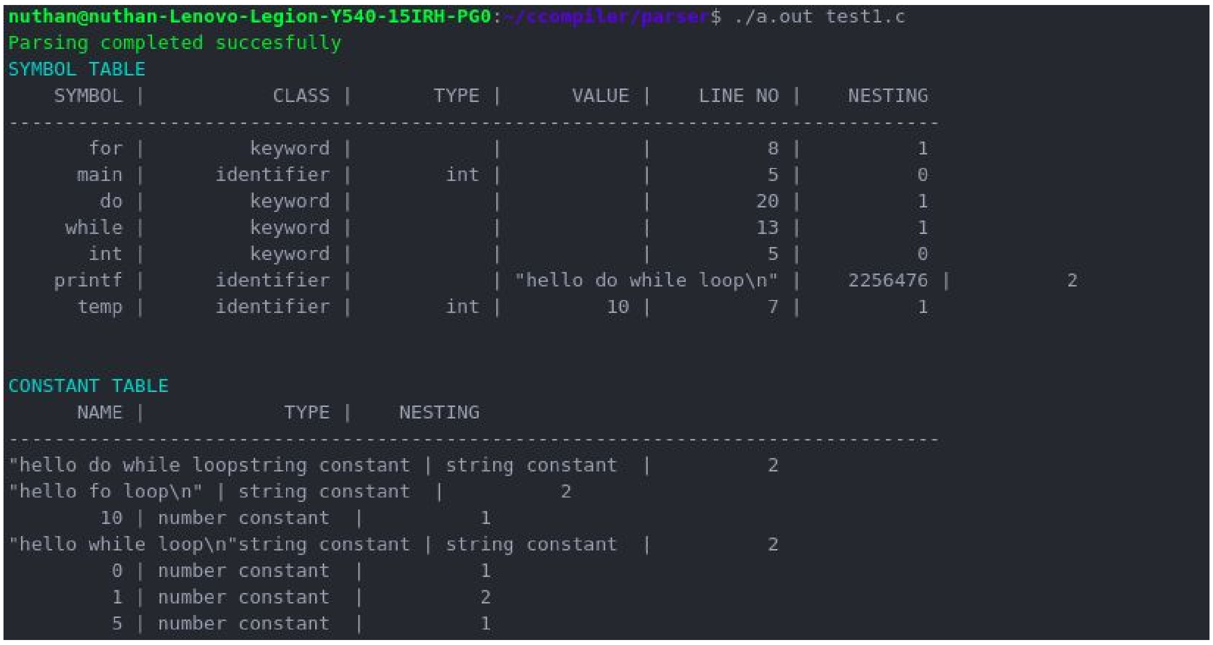
Test cases

Error free code

Test 1

```
// to work in all kind of loops for, while, do while

#include <stdio.h>

int main()
{
    int temp = 0;
    for (temp = 0; temp < 5; temp++)
    {
        printf("hello fo loop\n");
    }

    while (temp > 0)
    {

        printf("hello while loop\n");
        temp = temp - 1;
    }

    do
    {
        printf("hello do while loop\n");
        temp = temp - 1;
    } while (temp > 10);
}
```

Result



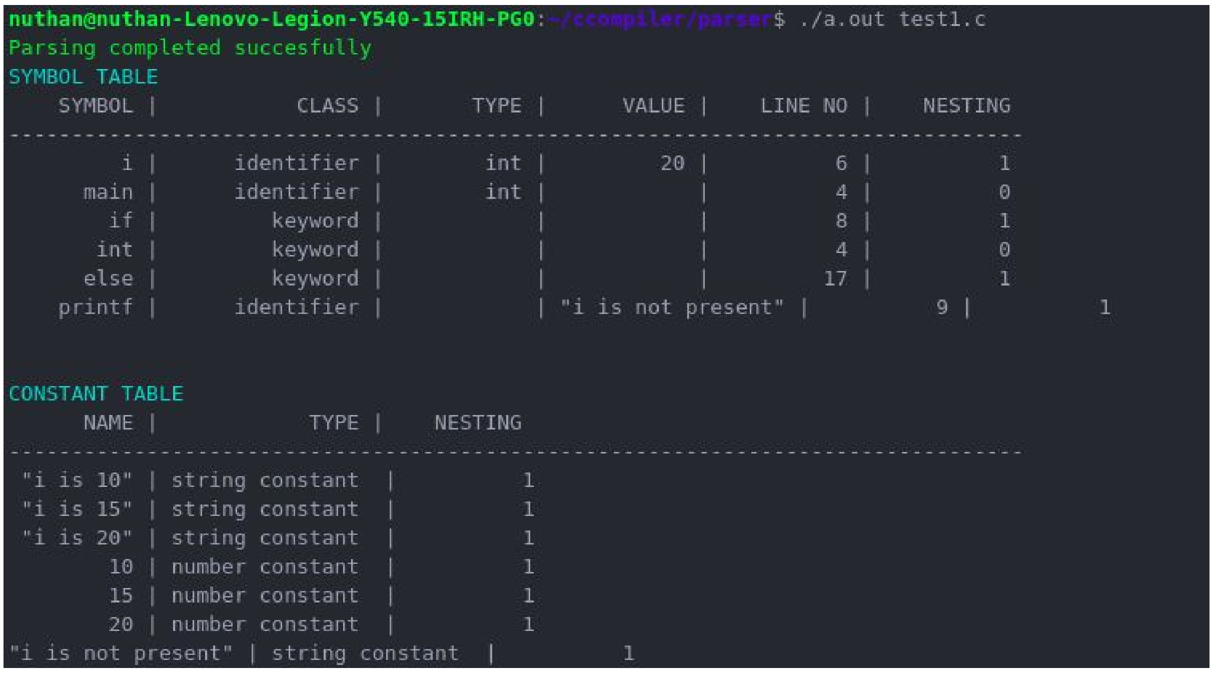
```
nuthan@nuthan-Lenovo-Legion-Y540-15IRH-PG0:~/ccompiler/parser$ ./a.out test1.c
Parsing completed succesfully
SYMBOL TABLE
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO |     NESTING
---------------------------------------------------------------------------------
       for |         keyword |            |            |          8 |           1
      main |      identifier |        int |            |          5 |           0
        do |         keyword |            |            |         20 |           1
     while |         keyword |            |            |         13 |           1
       int |         keyword |            |            |          5 |           0
    printf |      identifier |            | "hello do while loop\n" |     2256476 |           2
      temp |      identifier |        int |         10 |          7 |           1


CONSTANT TABLE
      NAME |            TYPE |     NESTING
---------------------------------------------------------------------------------
"hello do while loopstring constant | string constant  |           2
"hello fo loop\n" | string constant  |           2
        10 | number constant  |           1
"hello while loop\n"string constant | string constant  |           2
         0 | number constant  |           1
         1 | number constant  |           2
         5 | number constant  |           1
```

Figure 1: Output for Error free code, Test 1

Test 2

```c
//simple if else ladder

#include <stdio.h>
int main()
{
        int i = 20;

        if (i == 10)
                printf("i is 10");

        else if (i == 15)
                printf("i is 15");

        else if (i == 20)
                printf("i is 20");

        else
                printf("i is not present");
}
```

Result



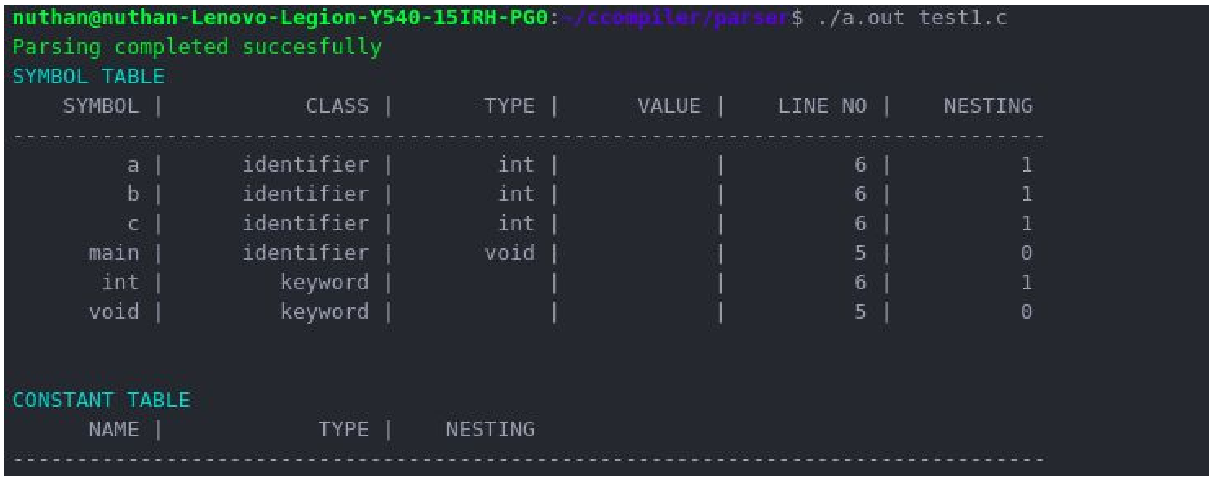
```
nuthan@nuthan-Lenovo-Legion-Y540-15IRH-PG0:~/ccompiler/parser$ ./a.out test1.c
Parsing completed succesfully
SYMBOL TABLE
    SYMBOL |            CLASS |        TYPE |       VALUE |     LINE NO |      NESTING
---------------------------------------------------------------------------------
         i |       identifier |         int |          20 |           6 |            1
      main |       identifier |         int |             |           4 |            0
        if |          keyword |             |             |           8 |            1
       int |          keyword |             |             |           4 |            0
      else |          keyword |             |             |          17 |            1
    printf |       identifier |             | "i is not present" |           9 |            1


CONSTANT TABLE
      NAME |            TYPE |     NESTING
---------------------------------------------------------------------------------
 "i is 10" | string constant  |          1
 "i is 15" | string constant  |          1
 "i is 20" | string constant  |          1
        10 | number constant  |          1
        15 | number constant  |          1
        20 | number constant  |          1
"i is not present" | string constant  |          1
```

Figure 2: Output for Error free code, Test 2

Test 3

```
#include<stdio.h>

//testing all operators

void main(){
       int a,b,c;


       c=a+b;
       c = a-b;
       c=a*b;
       c=a/b;
       c=a%b;
       c=a&&b;
       c=a||b;
       c=a*(a+b);
       c=a*a+b*b;


}
```

Result



```
nuthan@nuthan-Lenovo-Legion-Y540-15IRH-PG0:~/ccompiler/parser$ ./a.out test1.c
Parsing completed succesfully
SYMBOL TABLE
    SYMBOL |            CLASS |        TYPE |       VALUE |     LINE NO |     NESTING
---------------------------------------------------------------------------------
         a |       identifier |         int |             |           6 |           1
         b |       identifier |         int |             |           6 |           1
         c |       identifier |         int |             |           6 |           1
      main |       identifier |        void |             |           5 |           0
       int |          keyword |             |             |           6 |           1
      void |          keyword |             |             |           5 |           0


CONSTANT TABLE
      NAME |             TYPE |     NESTING
---------------------------------------------------------------------------------
```

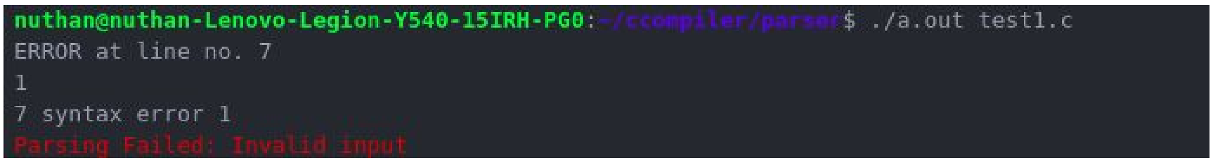
Figure 3: Output for Error free code, Test 3

Code with errors

Test 1

```
// invalid identifier rules

#include <stdio.h>

int main()
{
	int 123n = 2;
	printf("%d", 123n);
	123n --;
}
```

Result



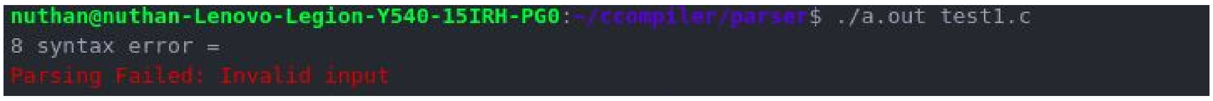
```
nuthan@nuthan-Lenovo-Legion-Y540-15IRH-PG0:~/ccompiler/parser$ ./a.out test1.c
ERROR at line no. 7
1
7 syntax error 1
Parsing Failed: Invalid input
```

Figure 4: Output for Code with errors, Test 1

Test 2

```c
// simple syntax error

#include <stdio.h>

void main()
{
    int a;
    a *a = a;
}
```

Result



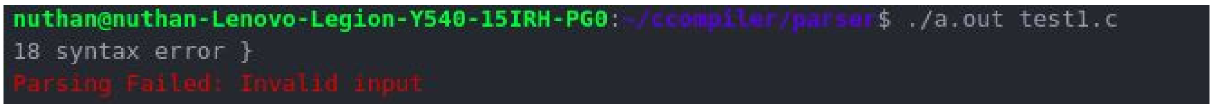
```
nuthan@nuthan-Lenovo-Legion-Y540-15IRH-PG0:~/ccompiler/parser$ ./a.out test1.c
8 syntax error =
Parsing Failed: Invalid input
```

Figure 5: Output for Code with errors, Test 2

Test 3

```c
//error test for function return mismatch
#include<stdio.h>

void Void_func_with_return(int a)
{
    int abc=1;
    return a;
}

void main()
{
    int i,n;

    Void_func_with_return(i);

    printf("%d",i)

}
```

Result



```
nuthan@nuthan-Lenovo-Legion-Y540-15IRH-PG0:~/ccompiler/parser$ ./a.out test1.c
18 syntax error }
Parsing Failed: Invalid input
```

Figure 6: Output for Code with errors, Test 3

Implementation

The previously-built lexer worked on the features of C using regular expressions while also taking into account the following corner cases:

1. The regular expression for identifiers
2. Support for multiline comments
3. Literals
4. Error handling for incomplete strings
5. Error handling for nested comments

The parser accompanying this report makes use of special production rules to implement C grammar and parse the tokens sourced from the lexer. It takes each token, parses it using the production rules and adds it to the symbol table along with the token's type, value and line of declaration. In the case of a parsing error, the line number and error are printed. The scanner code requires the functions mentioned in its explanation above to build the symbol table.

First table for nonterminals

Non-terminal	FIRST
code	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier, epsilon,
decl_list	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier, epsilon,
D	epsilon, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier, epsilon,
decl	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier, epsilon,
var_decl	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT
var_decl_list	identifier
V	‘,’,epsilon
var_decl_identifier	identifier
var_decl_identification	‘[‘, epsilon
identifier_array_type	‘[‘, epsilon
initialization_params	‘]’, integer_constant
initialization	epsilon, assignment_operator,
storage_class_specifier	AUTO, STATIC, EXTERN, REGISTER, epsilon
type_specifier	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED
type_spec	INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED
unsigned_grammar	Int, long, short, epsilon
signed_grammar	Int, long, short, epsilon
long_grammar	Int, epsilon
short_grammar	Int, epsilon
struct_decl	STRUCT
V1	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, identifier, epsilon
structure_decl	STRUCT
funct_decl	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, identifier, epsilon
funct_decl_type	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, identifier, epsilon
funct_decl_params	‘)’, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, identifier, epsilon
params	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, identifier, epsilon

parameters_list	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, identifier, epsilon
parameters_identifiers_list	identifier
parameters_identifier_list_parts	‘,’, epsilon
param_identifier	identifier
param_identifier_parts	‘[‘, epsilon
stmt	‘{‘, IF, WHILE, FOR, DO, CONTINUE, BREAK, RETURN, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, SIGNED, UNSIGNED, identifier, exclamation_operator, ‘(‘, ‘;’
compound_stmt	‘{‘
statment_list	‘{‘, IF, WHILE, FOR, DO, CONTINUE, BREAK, RETURN, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, SIGNED, UNSIGNED, identifier, exclamation_operator, ‘(‘, ‘;’
expression_stmt	‘;’, identifier, ‘(‘, exclamation_operator
conditional_stmt	IF
conditional_stmt_parts	ELSE, ';'
iterative_stmt	WHILE, FOR, DO
flow_stmt	CONTINUE, BREAK, RETURN,
string_initialization	assignment_operator
array_initialization	assignment_operator
array_int_decl	integer_constant
array_int_decl_parts	,', ε
expression	IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator
expression_parts	assignment_operator, addition_assignment_operator, subtraction_assignment_operator, multiplication_assignment_operator, division_assignment_operator, modulo_assignment_operaotor, increment_operator, decrement_operator
simple_expression	IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator
simple_expression_parts	OR_operator
and_expression	IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator
and_expression_parts	AND_operator,ε
unary_relation_expression	IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator
regular_expression	IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant
regular_expression_parts	geq_operator, leq_operator, ge_operator, le_operator, equality_operator, inequality_operator, ε
relational_operators	geq_operator, leq_operator, ge_operator, le_operator, equality_operator, inequality_operator

sum_expression	IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant
sum_operators	add_operator,subtract_operator
term	'(',identifier,multiplication_operator,division_operator,modulo_operator
Ax	multiplication_operator,division_operator,modulo_operator, epsilon
MULOP	multiplication_operator,division_operator,modulo_operator
factor	'(',identifier, integer_constant, string_constant, float_constant,character_constant
mutable	identifier
Bx	'[','.',epsilon
mutable_parts	'[','.'
immutable	'(',identifier, integer_constant, string_constant, float_constant,character_constant
call	identifier
arguments	expression, epsilon
arguments_list	expression
E	',',epsilon
constant	integer_constant, string_constant, float_constant,character_constant

Table 1: First table for non-terminals

Follow table for nonterminals

Non-terminal	FOLLOW
code	$
decl_list	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, IDENTIFIER,
D	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, IDENTIFIER,
decl	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, IDENTIFIER,
var_decl	‘;’, STRUCT, identifier
var_decl_list	‘;’, identifier, ‘]’, integer constant, ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ')'
V	‘;’, identifier, ‘]’, integer constant, ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ')'
var_decl_identifier	identifier, ‘]’, integer constant, ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ')'
var_decl_identification	‘]’, integer constant, ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ')'
identifier_array_type	‘]’, integer constant
initialization_params	assignment operator
initialization	assignment operator,
storage_class_specifier	AUTO, STATIC, EXTERN, REGISTER, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED
type_specifier	VOID, follow(type_spec)
type_spec	INT, CHAR, FLOAT, DOUBLE,
unsigned_grammar	INT, CHAR, FLOAT, DOUBLE,
signed_grammar	INT, CHAR, FLOAT, DOUBLE,
long_grammar	INT, CHAR, FLOAT, DOUBLE,
short_grammar	INT, CHAR, FLOAT, DOUBLE,
struct_decl	‘;’, STRUCT, identifier
V1	‘;’, STRUCT, identifier, ‘}’
structure_decl	‘;’, STRUCT, identifier
funct_decl	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier,
funct_decl_type	‘)’, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, identifier, epsilon
funct_decl_params	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier,
params	‘)’
parameters_list	‘)’

parameters_identifiers_list	‘)’
parameters_identifier_list_parts	‘)’
param_identifier	‘,’, ‘)’
param_identifier_parts	‘,’, ‘)’
stmt	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier,
compound_stmt	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier,
statment_list	‘}’
expression_stmt	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier,
conditional_stmt	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier,
conditional_stmt_parts	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier,
iterative_stmt	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier,
flow_stmt	AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, UNSIGNED, SIGNED, STRUCT, identifier,
string_initialization	assignment operator
array_initialization	assignment operator
array_int_decl	'}'
array_int_decl_parts	'}'
expression	‘]’, integer constant, ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ')'
expression_parts	‘]’, integer constant, ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ')'
simple_expression	‘]’, ')', ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ‘{‘, IF, WHILE, FOR, DO, CONTINUE, BREAK, RETURN, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, SIGNED, UNSIGNED, identifier, ELSE,
simple_expression_parts	‘]’, ')', ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ‘{‘, IF, WHILE, FOR, DO, CONTINUE, BREAK, RETURN, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, SIGNED, UNSIGNED, identifier, ELSE,
and_expression	OR_operator, ‘]’, ')', ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ‘{‘, IF, WHILE, FOR, DO, CONTINUE, BREAK, RETURN, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, SIGNED, UNSIGNED, identifier, ELSE,
and_expression_parts	OR_operator, ‘]’, ')', ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ‘{‘, IF, WHILE, FOR, DO, CONTINUE, BREAK, RETURN, AUTO, STATIC, EXTERN,

	REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, SIGNED, UNSIGNED, identifier, ELSE,
unary_relation_expression	AND_operator, OR_operator, ‘]’, ')', ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ‘{‘, IF, WHILE, FOR, DO, CONTINUE, BREAK, RETURN, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, SIGNED, UNSIGNED, identifier, ELSE,
regular_expression	AND_operator, OR_operator, ‘]’, ')', ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ‘{‘, IF, WHILE, FOR, DO, CONTINUE, BREAK, RETURN, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, SIGNED, UNSIGNED, identifier, ELSE,
regular_expression_parts	AND_operator, OR_operator, ‘]’, ')', ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ‘{‘, IF, WHILE, FOR, DO, CONTINUE, BREAK, RETURN, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, SIGNED, UNSIGNED, identifier, ELSE,
relational_operators	IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant
sum_expression	AND_operator, OR_operator, ‘]’, ')', ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ‘{‘, IF, WHILE, FOR, DO, CONTINUE, BREAK, RETURN, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, SIGNED, UNSIGNED, identifier, ELSE, geq_operator, leq_operator, ge_operator, le_operator, equality_operator, inequality_operator,
sum_operators	'(',identifier,multiplication_operator,division_operator,modulo_operator
term	AND_operator, OR_operator, ‘]’, ')', ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ‘{‘, IF, WHILE, FOR, DO, CONTINUE, BREAK, RETURN, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, SIGNED, UNSIGNED, identifier, ELSE, geq_operator, leq_operator, ge_operator, le_operator, equality_operator, inequality_operator,
A’	AND_operator, OR_operator, ‘]’, ')', ';', IDENTIFIER, '(', integer_constant, string_constant, float_constant, character_constant, exclaination_operator, ‘{‘, IF, WHILE, FOR, DO, CONTINUE, BREAK, RETURN, AUTO, STATIC, EXTERN, REGISTER, VOID, INT, CHAR, FLOAT, DOUBLE, LONG, SHORT, SIGNED, UNSIGNED, identifier, ELSE, geq_operator, leq_operator, ge_operator, le_operator, equality_operator, inequality_operator,
MULOP	'(',identifier, integer_constant, string_constant, float_constant,character_constant

factor	multiplication_operator,division_operator,modulo_operator, epsilon
mutable	multiplication_operator,division_operator,modulo_operator, epsilon, assignment_operator, addition_assignment_operator, subtraction_assignment_operator, multiplication_assignment_operator, division_assignment_operator, modulo_assignment_operaotor, increment_operator, decrement_operator
B’	multiplication_operator,division_operator,modulo_operator, epsilon, assignment_operator, addition_assignment_operator, subtraction_assignment_operator, multiplication_assignment_operator, division_assignment_operator, modulo_assignment_operaotor, increment_operator, decrement_operator
mutable_parts	'[','.',epsilon
immutable	multiplication_operator,division_operator,modulo_operator, epsilon
call	multiplication_operator,division_operator,modulo_operator, epsilon
arguments	‘}’
arguments_list	‘}’
E	‘}’
constant	multiplication_operator,division_operator,modulo_operator, epsilon

Table 2: Follow table for non-terminals

Limitations and future work

The parser is limited to a small subset of the full collection of syntactical rules associated with the C language. Therefore, more work is needed to increase the coverage of C’s collection of syntactical rules by the parser to make it fully appropriate for the C language.

References

(Red Dragon Book) - Aho A.V, Sethi R, and Ullman J.D. Compilers: Principles, Techniques, and
Tools. Addison-Wesley, 1986.

Grammar rules for C language:
http://www.quut.com/c/ANSI-C-grammar-y.html
http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf

Lex and Yacc Tutorial:
https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf

Course plan for list of specifications