Parser for the C Language

National Institute of Technology Karnataka Surathkal

Date: 7th February, 2019

Submitted To: Dr. Santhi Thilagam

Group Members:
Anshul Pinto 16CO101
Jay Satish Shinde 16CO118
Mohit Bhasi 16CO126

Abstract

This report contains the details of the tasks finished as a part of the Phase Two of Compilers Lab. We have developed a Parser for C language which makes use of the C lexer to parse the given C input file. The parser generates list of identifiers and functions with their types and also specifies syntax errors is any.

The parser code has functionality of taking input through a file or through standard input. This makes it more user friendly and efficient at the same time.

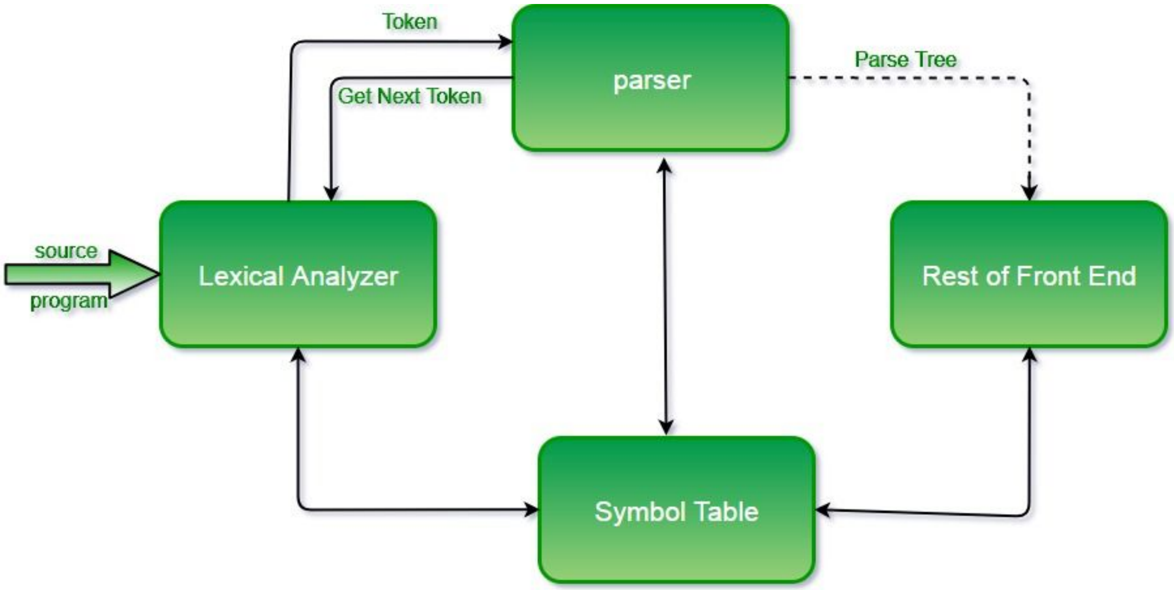
Contents

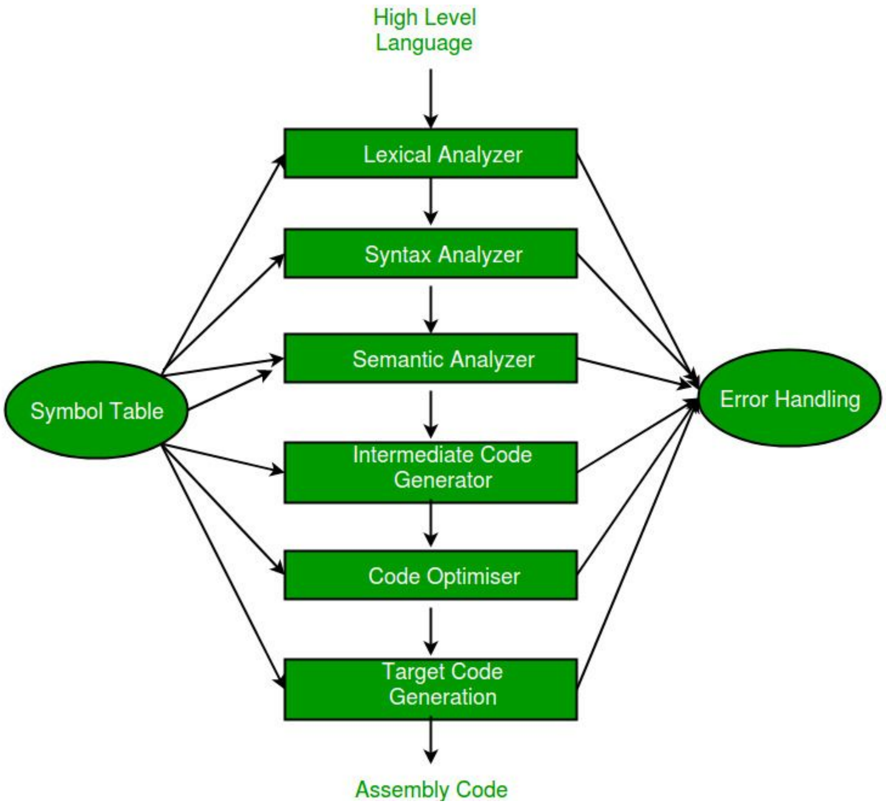
List of Figures and Tables:

Introduction

Parser/Syntactic Analysis

In the syntax analysis phase, 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.





Yacc Script

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.
The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.
Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.
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

%%
Subroutines

Input to yacc is divided into three sections. The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file. The definitions section consists of token declarations and C code bracketed by "**%{**" and "**%}**". The BNF grammar is placed in the rules section and user subroutines are added in the subroutines section.

C Program

This section describes the input C program which is fed to the yacc script for parsing. The workflow is explained as under:

- Compile the script using Yacc tool
  - $ yacc –d c_parser.y
- Compile the flex script using Flex tool
  - $ flex c_lexer.l
- After compiling the lex file, lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.
- The three files, lex.yy.c, y.tab.c and y.tab.h are compiled together with the options –ll and –ly
  - $ gcc –o compiler lex.yy.c y.tab.h y.tab.c –ll –ly
- The executable file is generated, which on running parses the C file given as a command line input
  - $ ./compiler test.c

The script also has an option to take standard input instead of taking input from a file.

Design of Programs

Updated Lexer Code

```
letter          [A-Za-z_]

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

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

%x comment string

%%

"/*"                   {beginning = yylineno; BEGIN comment;}
<comment>.|{whitespace}{}
<comment>"\n"         {yylineno++;}
<comment>"*/"         {BEGIN INITIAL;}
<comment>"/*"         {printf("Line %d: Nested comments are not
valid!\n",yylineno);}
<comment><<EOF>>    {printf("Line %d: Unterminated comment\n", beginning);
}

"//".*              {printf("Single line comment: %s \n",yytext);}

"#include<"({letter})*".h>" {}
"#define"({whitespace})""({letter})""({letter}|{digit})*""({whitespace})""(
{digit})+""                     {}
```

```
"#define"({whitespace})""({letter}({letter}|{digit})*)""({whitespace})""(({digit}+)\.({digit}+))""          {}
"#define"({whitespace})""({letter}({letter}|{digit})*)""({whitespace})""({letter}({letter}|{digit})*)""    {}

\"[^\n]*\"                    { yylval = yytext; return STRING_CONSTANT; }
\'{letter}\'                  { yylval = yytext; return CHAR_CONSTANT; }
{digit}+                      { yylval = yytext; return INT_CONSTANT; }
({digit}+)\.({digit}+)        { yylval = yytext; return FLOAT_CONSTANT; }
({digit}+)\.({digit}+)([eE][-+]?[0-9]+)? { yylval = yytext; return FLOAT_CONSTANT; }
[+\-]?[0][x|X]{hex}+          { yylval = yytext; return HEX_CONSTANT; }

"sizeof"        { return SIZEOF; }
"char"          { yylval = yytext; return CHAR; }
"short"         { yylval = yytext; return SHORT; }
"int"           { yylval = yytext; return INT; }
"long"          { yylval = yytext; return LONG; }
"signed"        { yylval = yytext; return SIGNED; }
"unsigned"      { yylval = yytext; return UNSIGNED; }
"void"          { yylval = yytext; return VOID; }
"if"            { return IF; }
"else"          { return ELSE; }
"while"         { return WHILE; }
"break"         { return BREAK; }
"return"        { return RETURN; }
"continue"      { return CONTINUE; }
"float"         { return FLOAT; }
"auto"          { return AUTO; }
"const"         { return CONST; }
"double"        { return DOUBLE; }
"extern"        { return EXTERN; }
"register"      { return REGISTER; }
"static"        { return STATIC; }
"inline"        { return INLINE; }
"typedef"       { return TYPEDEF; }
"case"          { return CASE; }
"switch"        { return SWITCH; }
```

```
"default"        { return DEFAULT; }
"do"             { return DO; }
"else if"        { return ELSE_IF; }
"for"            { return FOR; }
"goto"           { return GOTO; }

"++"             {  return INC_OP; }
"--"             {  return DEC_OP; }
"<="             {  return LE_OP; }
">="             {  return GE_OP; }
"=="             {  return EQ_OP; }
"!="             {  return NE_OP; }
"&&"             {  return AND_OP; }
"||"             {  return OR_OP; }
";"              {  return(';'); }
("{")            {  return('{'); }
("}")            {  return('}'); }
","              {  return(','); }
":"              {  return(':'); }
"="              {  return('='); }
"("              {  return('('); }
")"              {  return(')'); }
("["|"<:")       {  return('['); }
("]"|":>")       {  return(']'); }
"&"              {  return('&'); }
"-"              {  return('-'); }
"+"              {  return('+'); }
"*"              {  return('*'); }
"/"              {  return('/'); }
"%"              {  return('%'); }
"<"              {  return('<'); }
">"              {  return('>'); }
"^"              {  return('^'); }
"|"              {  return('|'); }
"?"              {  return('?'); }

{identifier} {
     if (strlen(yytext)>32)
```

```
          printf("Error: Identifier too long\n");
     else{
          printf("Identifier: %s\n",yytext );
          yylval = yytext;
          return IDENTIFIER;
     }
}


\n         { yylineno++; }
[ \t\v\f] {}
.          {}
%%
yywrap()
{
     return(1);
}
```

Parser Code

```
%nonassoc NO_ELSE
%nonassoc ELSE
%nonassoc ELSE_IF
%left '<' '>' '=' GE_OP LE_OP EQ_OP NE_OP
%left  '+'  '-'
%left  '*'  '/' '%'
%left  '|'
%left  '&'
%token IDENTIFIER STRING_CONSTANT CHAR_CONSTANT INT_CONSTANT
FLOAT_CONSTANT HEX_CONSTANT SIZEOF
%token INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP
```

```
%token TYPE_NAME DEF
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT VOID AUTO CONST DOUBLE
EXTERN REGISTER STATIC INLINE TYPEDEF
%token IF ELSE WHILE CONTINUE BREAK RETURN ELSE_IF GOTO DO FOR
%token CASE DEFAULT SWITCH
%start start_state
%nonassoc UNARY
%glr-parser

%{
#include<string.h>
#include "symboltable.h"
char type[100];
char temp[100];

entry_t** symbol_table;
entry_t** constant_table;
%}

%%

start_state
     : global_declaration
     | start_state global_declaration
     ;

global_declaration
     : function_definition
     | declaration
     ;

function_definition
     : declaration_specifiers declarator compound_statement
     | declarator compound_statement
     ;

fundamental_exp
```

```
     : IDENTIFIER
     | STRING_CONSTANT   { insert(constant_table, $1, "string"); }
     | HEX_CONSTANT      { insert(constant_table, $1, "hexadecimal"); }
     | CHAR_CONSTANT     { insert(constant_table, $1, "char"); }
     | FLOAT_CONSTANT    { insert(constant_table, $1, "float"); }
     | INT_CONSTANT      { insert(constant_table, $1, "int"); }
     | '(' expression ')'
     ;

secondary_exp
     : fundamental_exp
     | secondary_exp '[' expression ']'
     | secondary_exp '(' ')'
     | secondary_exp '(' arg_list ')'
     | secondary_exp INC_OP
     | secondary_exp DEC_OP
     ;

arg_list
     : assignment_expression
     | arg_list ',' assignment_expression
     ;

unary_expression
     : secondary_exp
     | INC_OP unary_expression
     | DEC_OP unary_expression
     | unary_operator typecast_exp
     ;

unary_operator
     : '*'
     | '+'
     | '-'
     ;

typecast_exp
     : unary_expression
     | '(' type_name ')' typecast_exp
     ;

multdivmod_exp
     : typecast_exp
     | multdivmod_exp '*' typecast_exp
     | multdivmod_exp '/' typecast_exp
     | multdivmod_exp '%' typecast_exp
     ;

addsub_exp
     : multdivmod_exp
     | addsub_exp '+' multdivmod_exp
     | addsub_exp '-' multdivmod_exp
     ;

relational_expression
     : addsub_exp
     | relational_expression '<' addsub_exp
     | relational_expression '>' addsub_exp
     | relational_expression LE_OP addsub_exp
     | relational_expression GE_OP addsub_exp
     ;

equality_expression
     : relational_expression
     | equality_expression EQ_OP relational_expression
     | equality_expression NE_OP relational_expression
     ;

and_expression
     : equality_expression
     | and_expression '&' equality_expression
     ;

exor_expression
     : and_expression
     | exor_expression '^' and_expression
     ;

unary_or_expression
     : exor_expression
     | unary_or_expression '|' exor_expression
     ;

logical_and_expression
     : unary_or_expression
     | logical_and_expression AND_OP unary_or_expression
     ;

logical_or_expression
     : logical_and_expression
     | logical_or_expression OR_OP logical_and_expression
     ;

conditional_expression
     : logical_or_expression
     | logical_or_expression '?' expression ':' conditional_expression
     ;

assignment_expression
     : conditional_expression
     | unary_expression '=' assignment_expression
     ;

expression
     : assignment_expression
     | expression ',' assignment_expression
     ;

constant_expression
     : conditional_expression
     ;

declaration
     : declaration_specifiers init_declarator_list ';'
     | error
     ;

declaration_specifiers
     : type_specifier                        { strcpy(type, $1); }
     | type_specifier declaration_specifiers { strcpy(temp, $1);
strcat(temp, " "); strcat(temp, type); strcpy(type, temp); }
     ;

init_declarator_list
     : init_declarator
     | init_declarator_list ',' init_declarator
     ;

init_declarator
     : declarator
     | declarator '=' init
     ;

type_specifier
     : VOID         { $$ = "void"; }
     | AUTO         { $$ = "auto"; }
     | TYPEDEF      { $$ = "typedef"; }
     | EXTERN       { $$ = "extern"; }
     | REGISTER     { $$ = "register"; }
     | STATIC       { $$ = "static"; }
     | CHAR         { $$ = "char"; }
     | SHORT        { $$ = "short"; }
     | CONST        { $$ = "const"; }
     | FLOAT        { $$ = "float"; }
     | DOUBLE       { $$ = "double"; }
     | INT          { $$ = "int"; }
     | INLINE       { $$ = "inline"; }
     | LONG         { $$ = "long"; }
     | SIGNED       { $$ = "signed"; }
     | UNSIGNED     { $$ = "unsigned"; }
     ;
```

```
type_specifier_list
      : type_specifier type_specifier_list
      | type_specifier
      ;

declarator
      : IDENTIFIER    { insert(symbol_table, $1, type); }
      | '(' declarator ')'
      | declarator '[' constant_expression ']'
      | declarator '[' ']'
      | declarator '(' parameter_type_list ')'
      | declarator '(' identifier_list ')'
      | declarator '(' ')'
      ;


parameter_type_list
      : parameter_list
      ;

parameter_list
      : parameter_declaration
      | parameter_list ',' parameter_declaration
      ;

parameter_declaration
      : declaration_specifiers declarator
      | declaration_specifiers abstract_declarator
      | declaration_specifiers
      ;

identifier_list
      : IDENTIFIER
      | identifier_list ',' IDENTIFIER
      ;

type_name
      : type_specifier_list
      | type_specifier_list abstract_declarator
      ;

abstract_declarator
      : direct_abstract_declarator
      ;

direct_abstract_declarator
      : '(' abstract_declarator ')'
      | '[' ']'
      | '[' constant_expression ']'
      | direct_abstract_declarator '[' ']'
      | direct_abstract_declarator '[' constant_expression ']'
      | '(' ')'
      | '(' parameter_type_list ')'
      | direct_abstract_declarator '(' ')'
      | direct_abstract_declarator '(' parameter_type_list ')'
      ;

init
      : assignment_expression
      | '{' init_list '}'
      | '{' init_list ',' '}'
      ;

init_list
      : init
      | init_list ',' init
      ;

statement
      : compound_statement
      | expression_statement
      | selection_statement
      | iteration_statement
      | jump_statement
      | case_statement
      ;
```

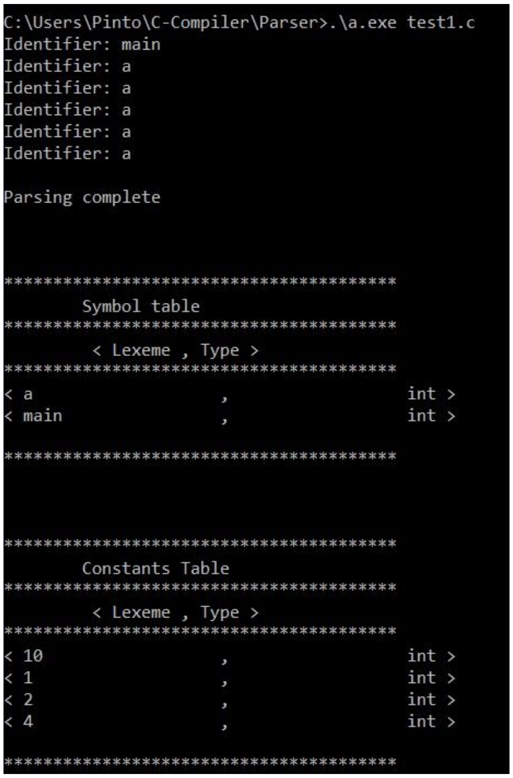
```
compound_statement
      : '{' '}'
      | '{' statement_list '}'
      | '{' declaration_list '}'
      | '{' declaration_list statement_list '}'
      | '{' declaration_list statement_list declaration_list statement_list '}'
      | '{' declaration_list statement_list declaration_list '}'
      | '{' statement_list declaration_list statement_list '}'
      ;

declaration_list
      : declaration
      | declaration_list declaration
      ;

statement_list
      : statement
      | statement_list statement
      ;

expression_statement
      : ';'
      | expression ';'
      ;

else_list
      : ELSE_IF '(' expression ')' statement else_list
      | ELSE statement
      ;

case_statement
      : CASE CHAR_CONSTANT ':' statement
      | CASE INT_CONSTANT ':' statement
      | DEFAULT ':'
      ;
```

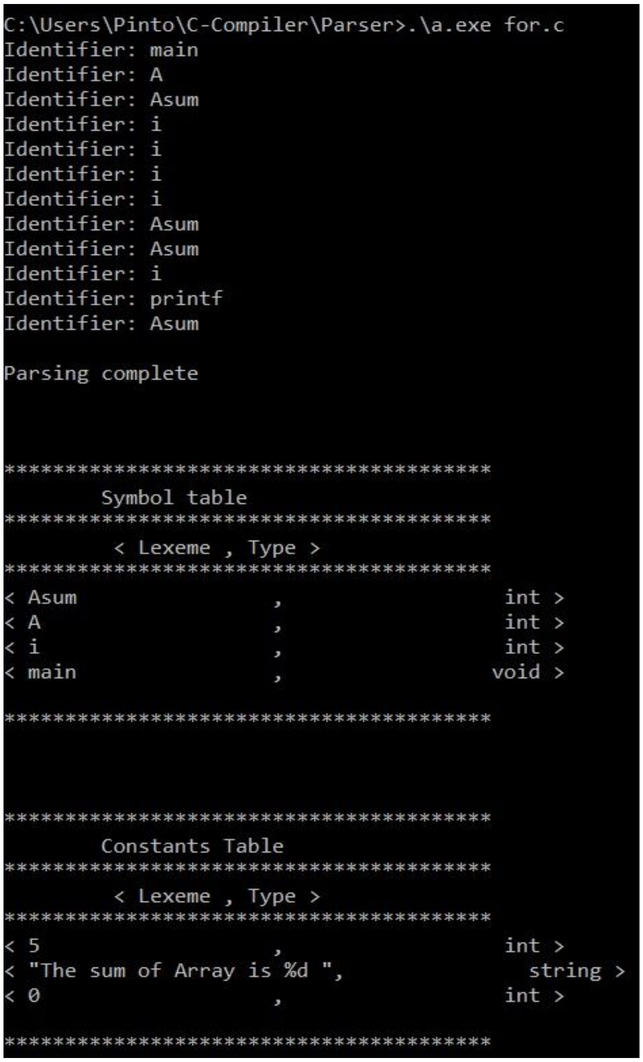
```
selection_statement
     : IF '(' expression ')' statement %prec NO_ELSE
     | IF '(' expression ')' statement else_list
     | SWITCH '(' IDENTIFIER ')' statement
     ;

iteration_statement
     : WHILE '(' expression ')' statement
     | FOR '(' expression ';' expression ';' expression ')' statement
     | DO statement WHILE '(' expression ')' ';'
     ;

jump_statement
     : CONTINUE ';'
     | BREAK ';'
     | RETURN ';'
     | RETURN expression ';'
     | GOTO IDENTIFIER ':'
     ;
%%
#include"lex.yy.c"
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int err=0;
int main(int argc, char *argv[])
{
     symbol_table=create_table();
     constant_table=create_table();
     yyin = fopen(argv[1], "r");
     yyparse();
     if(err==0)
          printf("\nParsing complete\n");
     else
          printf("\nParsing failed\n");
     fclose(yyin);
     printf("\n\n");
     printf("\n****************************************");
     printf("\n\tSymbol table");
     display(symbol_table);
     printf("\n\n");
     printf("\n****************************************");
     printf("\n\tConstants Table");
     display(constant_table);
     printf("\n\n");
     return 0;
}
extern char *yytext;
yyerror(char *s)
{
     err=1;
     printf("\nLine %d : %s\n", (yylineno), s);
     printf("\n\n");
     printf("\n****************************************");
     printf("\n\tSymbol table");
     display(symbol_table);
     printf("\n\n");
     printf("\n****************************************");
     printf("\n\tConstants Table");
     display(constant_table);
     printf("\n\n");
     exit(0);
}
```

Test Cases

Without Errors:

S NO	Test Case	Expected Output	Status

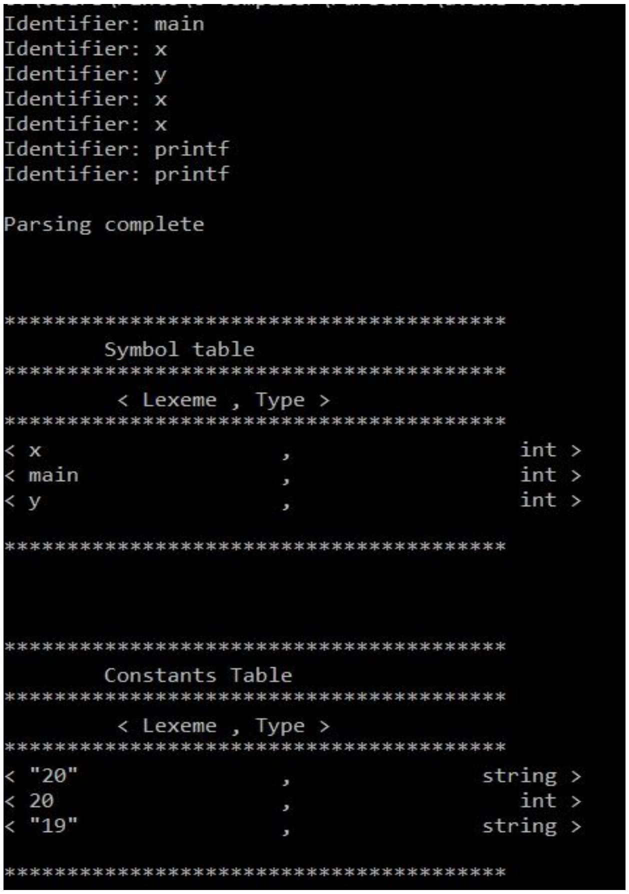
1	#include<stdio.h>
	int main()
	{
	    int a=4 ;
	     if(a==10)
	    {
	        a=a+2;
	    }
	    else if(1){
	    }
	    else if(1){
	    }
	    else{
	        a+1;
	    }
	}
		
		C:\Users\Pinto\C-Compiler\Parser>.\a.exe test1.c
		Identifier: main
		Identifier: a
		Identifier: a
		Identifier: a
		Identifier: a
		Identifier: a
		
		Parsing complete
		
		**
		        Symbol table
		**
		         < Lexeme , Type >
		**
		< a , int >
		< main , int >
		
		**
		
		**
		        Constants Table
		**
		         < Lexeme , Type >
		**
		< 10 , int >
		< 1 , int >
		< 2 , int >
		< 4 , int >
		
		**
			PASS

2	#include<stdio.h>
	void main()
	{
	    int A = 5;
	    int Asum = 0;
	    int i;
	    for (i=0; i<5;i++)
	    {
	        Asum = Asum + i;
	    }
	    printf("The sum of Array is %d ",Asum);
	}
		
		C:\Users\Pinto\C-Compiler\Parser>.\a.exe for.c
		Identifier: main
		Identifier: A
		Identifier: Asum
		Identifier: i
		Identifier: i
		Identifier: i
		Identifier: i
		Identifier: Asum
		Identifier: Asum
		Identifier: i
		Identifier: printf
		Identifier: Asum
		
		Parsing complete
		
		***
		        Symbol table
		***
		         < Lexeme , Type >
		***
		< Asum , int >
		< A , int >
		< i , int >
		< main , void >
		
		***
		
		***
		        Constants Table
		***
		         < Lexeme , Type >
		***
		< 5 , int >
		< "The sum of Array is %d ", string >
		< 0 , int >
		
		***
			PASS

<table>
<tr>
<td>3</td>
<td>

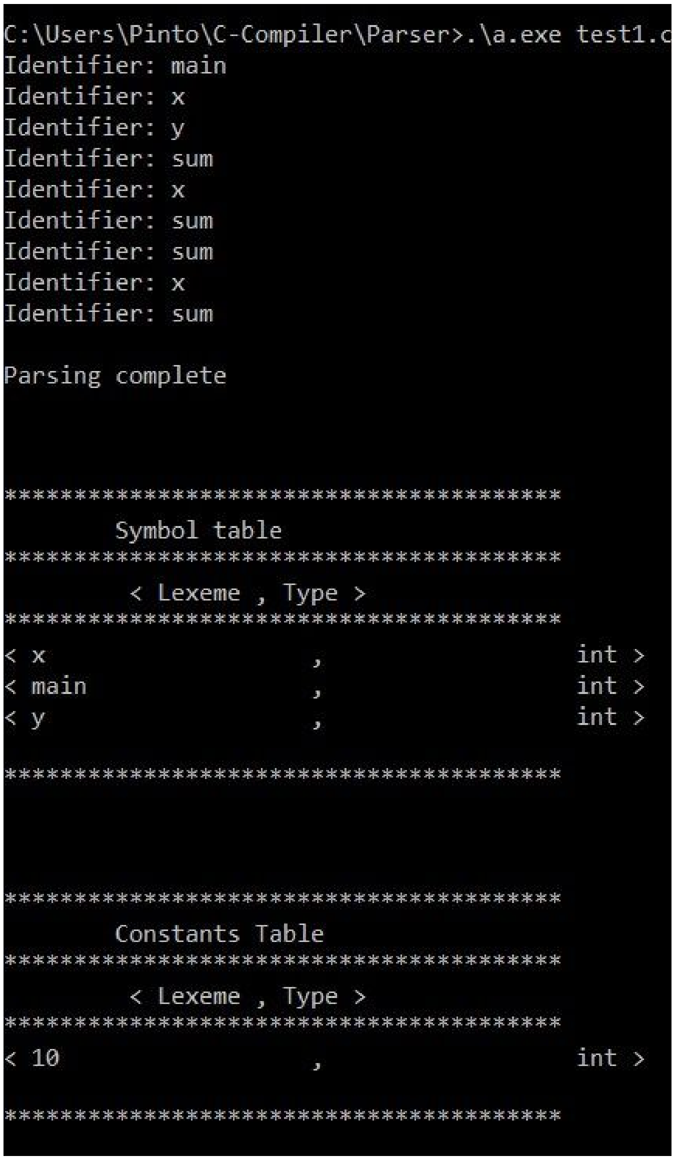
```
#include<stdio.h>

int main(void)
{
    int x , y;
    x = 20 ;
    switch(x)
    {
        case 19:
            printf("19");
            break;
        case 20:
            printf("20");
    }
}
```

</td>
<td>



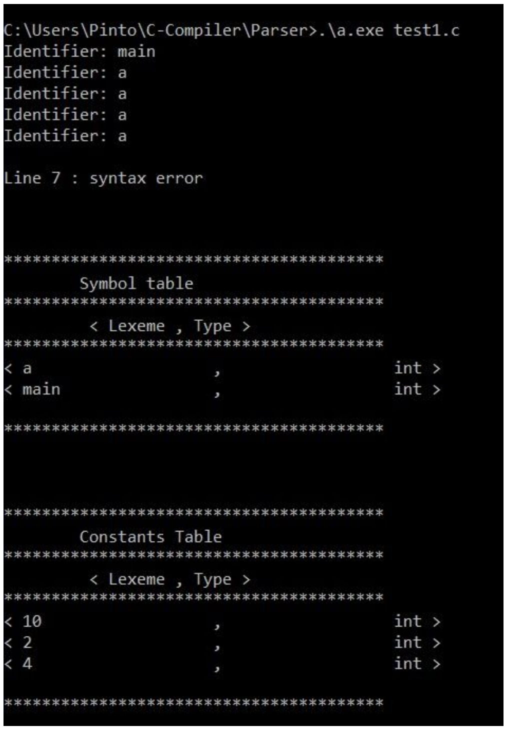
```
Identifier: main
Identifier: x
Identifier: y
Identifier: x
Identifier: x
Identifier: printf
Identifier: printf

Parsing complete



****************************************
        Symbol table
****************************************
         < Lexeme , Type >
****************************************
< x                   ,              int >
< main                ,              int >
< y                   ,              int >

****************************************



****************************************
        Constants Table
****************************************
         < Lexeme , Type >
****************************************
< "20"                ,           string >
< 20                  ,              int >
< "19"                ,           string >

****************************************
```

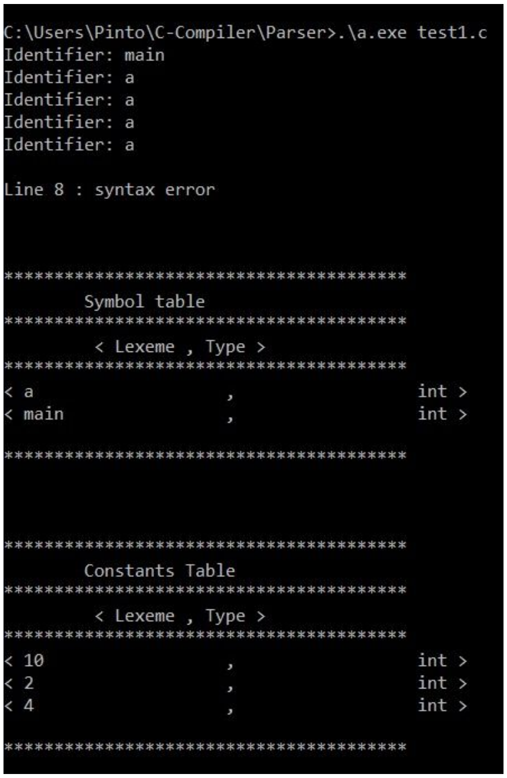
</td>
<td>PASS</td>
</tr>
</table>

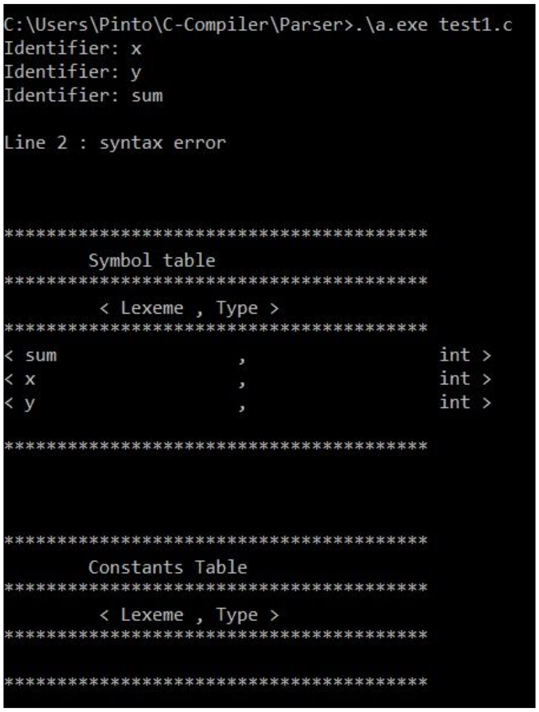
4	#include<stdio.h>
	
	int main(void)
	{
	int x , y;
	sum = -10;
	x = 10;
	do {
	sum = sum + x
	}
	while(sum<10);
	}
		
		C:\Users\Pinto\C-Compiler\Parser>.\a.exe test1.c
		Identifier: main
		Identifier: x
		Identifier: y
		Identifier: sum
		Identifier: x
		Identifier: sum
		Identifier: sum
		Identifier: x
		Identifier: sum
		
		Parsing complete
		
		**
		Symbol table
		**
		< Lexeme , Type >
		**
		< x , int >
		< main , int >
		< y , int >
		**
		
		**
		Constants Table
		**
		< Lexeme , Type >
		**
		< 10 , int >
		**
			PASS

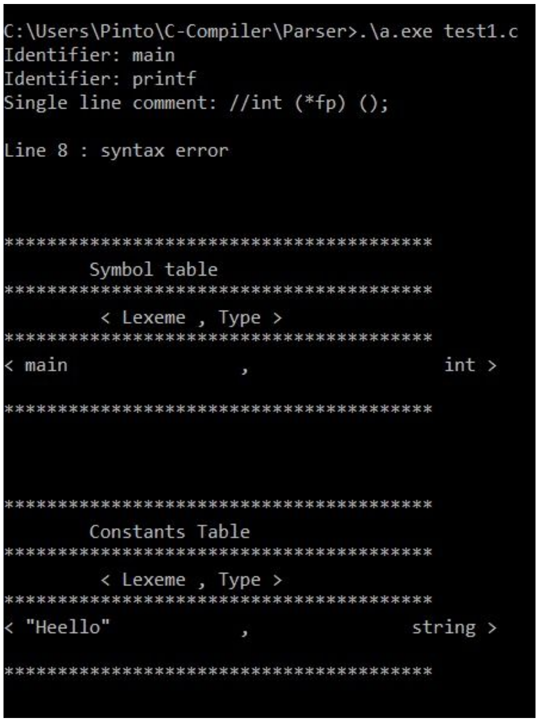
With Errors:

Serial no	Test Case	Output	Status

1	#include<stdio.h>
	int main()
	{
	int a=4 ;
	if(a==10)
	{
	a=a+2
	}
	else if(a==1)
	{
	a=a+3;
	}
	else{
	a+1;
	}
	}
		
		C:\Users\Pinto\C-Compiler\Parser>.\a.exe test1.c
		Identifier: main
		Identifier: a
		Identifier: a
		Identifier: a
		Identifier: a
		
		Line 7 : syntax error
		
		**
		Symbol table
		**
		< Lexeme , Type >
		**
		< a , int >
		< main , int >
		
		**
		
		**
		Constants Table
		**
		< Lexeme , Type >
		**
		< 10 , int >
		< 2 , int >
		< 4 , int >
		
		**
			FAIL

2	#include<stdio.h>
	int main()
	{
	int a=4 ;
	if(a==10)
	{
	a=a+2;
	
	else if(a==1)
	{
	a=a+3;
	}
	else{
	a+1;
	}
	}
		
		C:\Users\Pinto\C-Compiler\Parser>.\a.exe test1.c
		Identifier: main
		Identifier: a
		Identifier: a
		Identifier: a
		Identifier: a
		
		Line 8 : syntax error
		
		**
		Symbol table
		**
		< Lexeme , Type >
		**
		< a , int >
		< main , int >
		
		**
		
		**
		Constants Table
		**
		< Lexeme , Type >
		**
		< 10 , int >
		< 2 , int >
		< 4 , int >
		
		**
			FAIL

3	#include<stdio.h>
	int x , y;
	sum = -10;
	x = 10;
	do {
	sum = sum + x
	}
	while(sum<10);
		
		C:\Users\Pinto\C-Compiler\Parser>.\a.exe test1.c
		Identifier: x
		Identifier: y
		Identifier: sum
		
		Line 2 : syntax error
		
		**
		Symbol table
		**
		< Lexeme , Type >
		**
		< sum , int >
		< x , int >
		< y , int >
		
		**
		
		**
		Constants Table
		**
		< Lexeme , Type >
		**
		
		**
			FAIL

4	#include<stdio.h>
	
	int main()
	{
	printf("Heello");
	//int (*fp) ();
	}
	
	void (*(f[10]) (int, int)
		
		C:\Users\Pinto\C-Compiler\Parser>.\a.exe test1.c
		Identifier: main
		Identifier: printf
		Single line comment: //int (*fp) ();
		
		Line 8 : syntax error
		
		**
		Symbol table
		**
		< Lexeme , Type >
		**
		< main , int >
		
		**
		
		**
		Constants Table
		**
		< Lexeme , Type >
		**
		< "Heello" , string >
		
		**
			FAIL

Implementation

The project contains mainly a yacc file and a lex file. The lex file is modified so that it can be integrated into this phase. We have explained the grammar for the main constructs of the language below.
The yacc file initially contains all the token declarations.
Here we also specify the precedence of all the operators. It is in the increasing order of precedence. Next we start with the grammar for accepting various constructs for the C language.
We define start_state as the starting state for the grammar. Whenever we come across a constant of any type (int, hex, float) we add it to the constants table. declarator adds identifiers into the symbol table. We have grammar that accepts all arithmetic expressions using multdivmod_exp and addsub_exp.
selection_statement handles switch and if else statements.
iteration_statement handles all looping constructs such as while for and do-while.
jump_statement handle continue, break, return, go to,etc. case_statement takes care of syntax in a case statement of a switch.

Future work

The yacc script presented in this report takes care of all the rules of C language, but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle all aspects of C language and making it more efficient.

References

- **Compilers: Principles, Techniques, and Tools by Aho, Lam, Ullman, Sethi**
- **https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm**
- **https://www.geeksforgeeks.org/parsing-set-2-bottom-up-or-shift-reduce-parsers/**
- **http://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf**