Parser for the C Language

National Institute of Technology Karnataka,Surathkal

Date: 7th Feb 2019

Submitted To

Dr. P.Santhi Thilagam

Dept. of Computer Science

NITK Surathkal

Group Members:

1. Hardik Rana (16CO138)
2. Shushant Kumar (16CO143)
3. Anmol Horo (16CO206)

Abstract

This report contains the details of the tasks finished as a part of the Phase Two of Compiler design Lab. We have developed a Parser for C language which makes use of the C lexer to parse the given C input file. The parser 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

1. Introduction...2
   - Parser/Syntactic Analysis.......................................2
   - Yacc Script...3
   - C Program..3
2. Design of Programs...4
   - Code ....
   - Explanation ....
3. Test Cases 19
   - Without Errors...
   - With Errors ....
4. Implementation 31
5. Results/Future Work..33
6. References 33

1. Introduction

When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation. The aim of this phase of a compiler is to implement a parser for C language. The lexical analyzer generated in the first phase reads the source program and generates tokens that are given as input to the parser which then creates a syntax tree in accordance with the grammar, consequently leading to the generation of intermediate code that is fed into the synthesis phase, to obtain the correct, equivalent machine level code. We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by pushdown automata.

Parser/Syntactic Analysis:

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.

Syntax Analysis or **Parsing** is the second phase, i.e. after lexical analysis. A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. It hecks the syntactic structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a **Parse tree** or **Syntax tree** . The parse tree is constructed by using the predefined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. The Grammar for a Language consists of Production rules.

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

%%

C code section

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 rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

C Program:

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

1.Compile the script using Yacc tool

- **$ yacc –d parser.y -v**

2. Compile the flex script using Flex tool

- **$ lex lexl.l**

3. 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.

4. The three files, lex.yy.c, y.tab.c and y.tab.h are compiled together with the options –ll and –ly

- **$ gcc –w -g y.tab.c -ly -ll -o parser**

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

- **$ ./compiler test.c**

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

2. Design Of Programs

Code:

The entire code for lexical analysis is broken down into 3 files: scanner.l, token_number.h and table.h.

File	Contents
lex.l	A lex file containing the lex specification of regular expressions.
symboltable.h	Contains the definition of the symbol table and the constants table and also defines functions for inserting into the hash table and displaying its contents.
parser.y	

Lex Code(lex.l file)

```
%{

#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
/* #include "symboltable.h" */
#include "y.tab.h"

/* entry_t** symbol_table; */
/* entry_t** constant_table; */
int cmnt_strt = 0;

%}

letter [a-zA-Z]
digit [0-9]
ws  [ \t\r\f\v]+
und [_]
identifier (_|{letter})({letter}|{digit}|_){0,31}
hex [0-9a-f]

 /* Exclusive states */
```

```
%x CMNT
%x PREIN
%x PREDEF

%%
  /* Keywords*/


"main(void)" {}
"main()"  {}
"main(int argc, char **argv)" {}
"main(int argc, char *argv[])" {}

"int"                        {return INT;}
"float"                        {return FLOAT;}
"char"                        {return CHAR;}
"void"                        {return VOID;}
"long"                        {return LONG;}
"long long"                    {return LONG_LONG;}
"short"                        {return SHORT;}
"signed"                        {return SIGNED;}
"unsigned"                    {return UNSIGNED;}
"for"                        {return FOR;}
"while"                        {return WHILE;}
"break"                        {return BREAK;}
"continue"                    {return CONTINUE;}
"if"                        {return IF;}
"else"                        {return ELSE;}
"return"                        {return RETURN;}

{identifier}                    {yylval.entry = insert(symbol_table, yytext, INT_MAX); return
IDENTIFIER;}
{ws}                        ;
[+\-]?{digit}*\.{digit}+            {yylval.dval=atof(yytext);
                  insert( constant_table,yytext,FLOAT_CONSTANT );return FLOAT_CONSTANT;}
[+\-]?[0][x|X]{hex}+[lLuU]? {yylval.dval=(int)strtol(yytext,NULL,16);
                     insert( constant_table,yytext,HEX_CONSTANT);return  HEX_CONSTANT;}
[+\-]?{digit}+[lLuU]?                                    {yylval.dval=atoi(yytext);insert(
constant_table,yytext,DEC_CONSTANT); return  DEC_CONSTANT;}


"#ifdef"                                    {}
"#ifndef"                                    {}
"#if"                                        {}
"#else"                                        {}
"#elif"                                        {}
"#endif"                                        {}
```

```
"#error"                                              {}
"#pragma"                                             {}

"/*"                    {cmnt_strt = yylineno; BEGIN CMNT;}
<CMNT>.|{ws}                   ;
<CMNT>\n                 {yylineno++;}
<CMNT>"*/"               {BEGIN INITIAL;}
<CMNT>"/*"               {printf("Line %3d: Nested comments are not valid!\n",yylineno);}
<CMNT><<EOF>>                    {printf("Line %3d: Unterminated comment\n", cmnt_strt);
yyterminate();}
/* ^"#include"                {BEGIN PREIN;}
<PREIN>"<"[^<>\n]+">"           {printf("\t%-10s : %3d\n",yytext,HEADER_FILE);}
<PREIN>{ws}                  ;
<PREIN>\"[^"\n]+\"              {printf("\t%-10s : %3d\n",yytext,HEADER_FILE);}
<PREIN>\n                       {yylineno++; BEGIN INITIAL;}
<PREIN>.                        {printf("Line %3d: Illegal header file format \n",yylineno);}

^"#define"                             {BEGIN PREDEF;}
<PREDEF>{ws}+{letter}({letter}|{digit}|{und})*{ws}+{digit}+
                                       {printf("\t%-10s : %3d\n",yytext,DEFINE_FILE);}
<PREDEF>{ws}+{letter}({letter}|{digit}|{und})*{ws}+({digit}+)\.({digit}+)
                                       {printf("\t%-10s : %3d\n",yytext,DEFINE_FILE);}
<PREDEF>{ws}+{letter}({letter}|{digit}|{und})*{ws}+{letter}({letter}|{digit}|{und})*
                                        {printf("\t%-10s : %3d\n",yytext,DEFINE_FILE);}
<PREDEF>{ws}[^{ws}\n]+{ws}[^{ws}\n]+    {printf("\t%-10s : %3d\n",yytext,DEFINE_FILE);}
<PREDEF>\n                              {yylineno++; BEGIN INITIAL;}*/
<PREDEF>.
"//".*                    ;

\"[^\"\n]*\"     {

  if(yytext[yyleng-2]=='\\') /* check if it was an escaped quote */
  {
    yyless(yyleng-1);       /* push the quote back if it was escaped */
    yymore();
  }
  else{
  insert( constant_table, yytext, INT_MAX);
  return STRING;
  }
 }

 \'{letter}\'    {


  insert( constant_table, yytext, INT_MAX);
  return STRING_CONSTANT;
 }

\"[^\"\n]*$                    {printf("Line %3d: Unterminated string %s\n",yylineno,yytext);}
{digit}+({letter}|_)+           {printf("Line %3d: Illegal identifier name %s\n",yylineno,yytext);}
\n                          {yylineno++;}

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


"&&"                                        {return LOGICAL_AND;}
"||"                                        {return LOGICAL_OR;}
"<="                                        {return LS_EQ;}
">="                                        {return GR_EQ;}
"=="                                        {return EQ;}
"!="                                  {return NOT_EQ;}
"="                          { return '='; }
.                          {return yytext[0];}

%%
/*
int main()
{
  yyin=fopen("test-case-1.c","r");

  constant_table=create_table();
  symbol_table = create_table();

  yylex();

  printf("\n\tSymbol table");
  display(symbol_table);
  printf("\n\tConstants Table");
  display(constant_table);

  printf("NOTE: Please refer tokens.h for token meanings\n");
} */
```

symboltable.h file

```
/*
 File : symboltable .h
 Description : This file contains functions related to a hash organised
                        symbol table.
 The functions implemented are:
  create_table (), insert (), search , display ()
*/

#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <string.h>
#include <math.h>

#define HASH_TABLE_SIZE 100

/* struct to hold each entry */
struct entry_s
{
	char* lexeme;
	double value;
	int data_type;
	struct entry_s* successor;
};

typedef struct entry_s entry_t;

/* Create a new hash_table. */
entry_t** create_table()
{
	entry_t** hash_table_ptr = NULL; // declare a pointer

	/* Allocate memory for a hashtable array of size HASH_TABLE_SIZE */
	if( ( hash_table_ptr = malloc( sizeof( entry_t* ) * HASH_TABLE_SIZE ) ) == NULL )
	return NULL;

	int i;

	// Intitialise all entries as NULL
    for( i = 0; i < HASH_TABLE_SIZE; i++ )
	{
		hash_table_ptr[i] = NULL;
	}

	return hash_table_ptr;
}

/* Generate hash from a string. Then generate an index in [0, HASH_TABLE_SIZE) */
uint32_t hash( char *lexeme )
{
```

```
            size_t i;
            uint32_t hash;

            /* Apply jenkin's hash function
            * https://en.wikipedia.org/wiki/Jenkins_hash_function#one-at-a-time
            */
            for ( hash = i = 0; i < strlen(lexeme); ++i ) {
        hash += lexeme[i];
        hash += ( hash << 10 );
        hash ^= ( hash >> 6 );
    }
            hash += ( hash << 3 );
            hash ^= ( hash >> 11 );
    hash += ( hash << 15 );

            return hash % HASH_TABLE_SIZE; // return an index in [0, HASH_TABLE_SIZE)
}

/* Create an entry for a lexeme, token pair. This will be called from the insert function */
entry_t *create_entry( char *lexeme, double value )
{
            entry_t *newentry;

            /* Allocate space for newentry */
            if( ( newentry = malloc( sizeof( entry_t ) ) ) == NULL ) {
                        return NULL;
            }
            /* Copy lexeme to newentry location using strdup (string-duplicate). Return NULL if it fails
*/
            if( ( newentry->lexeme = strdup( lexeme ) ) == NULL ) {
                        return NULL;
            }

            newentry->value = value;
            newentry->successor = NULL;

            return newentry;
}

/* Search for an entry given a lexeme. Return a pointer to the entry of the lexeme exists, else
return NULL */
entry_t* search( entry_t** hash_table_ptr, char* lexeme )
{
            uint32_t idx = 0;
            entry_t* myentry;

    // get the index of this lexeme as per the hash function
            idx = hash( lexeme );

            /* Traverse the linked list at this idx and see if lexeme exists */
            myentry = hash_table_ptr[idx];

            while( myentry != NULL && strcmp( lexeme, myentry->lexeme ) != 0 )
        {
                myentry = myentry->successor;
        }

        if(myentry == NULL) // lexeme is not found
                return NULL;

        else // lexeme found
                return myentry;

}

/* Insert an entry into a hash table. */
entry_t* insert( entry_t** hash_table_ptr, char* lexeme, double value )
{
        entry_t* finder = search( hash_table_ptr, lexeme );
        if( finder != NULL) // If lexeme already exists, don't insert, return
           return finder ;

        uint32_t idx;
        entry_t* newentry = NULL;
        entry_t* head = NULL;

        idx = hash( lexeme ); // Get the index for this lexeme based on the hash function
        newentry = create_entry( lexeme, value ); // Create an entry using the <lexeme, token> pair

        if(newentry == NULL) // In case there was some error while executing create_entry()
        {
                printf("Insert failed. New entry could not be created.");
                exit(1);
        }

        head = hash_table_ptr[idx]; // get the head entry at this index

        if(head == NULL) // This is the first lexeme that matches this hash index
        {
                hash_table_ptr[idx] = newentry;
        }
        else // if not, add this entry to the head
        {
                newentry->successor = hash_table_ptr[idx];
                hash_table_ptr[idx] = newentry;
        }
        return hash_table_ptr[idx];
}

// Traverse the hash table and print all the entries
void display(entry_t** hash_table_ptr)
{
        int i;
        entry_t* traverser;
   printf("\n====================================================\n");
   printf(" %-20s %-20s %-20s\n","lexeme","value","data-type");
```

```
    printf("=========================================================\n");

            for( i=0; i < HASH_TABLE_SIZE; i++)
            {
                        traverser = hash_table_ptr[i];

                        while( traverser != NULL)
                        {
                                    if(traverser->value-(int)traverser->value){
                                                printf("     %-20s      %-20lf      %-20d       \n",      traverser->lexeme,
(double)traverser->value, traverser->data_type);
                                                traverser = traverser->successor;
                                    }
                                    else{
                                                printf("     %-20s      %-20d      %-20d       \n",      traverser->lexeme,
(int)traverser->value, traverser->data_type);
                                                traverser = traverser->successor;
                                    }
                        }
            }
    printf("=========================================================\n");

}
```

parser.y file

```
%{
    #include <stdlib.h>
    #include <stdio.h>
    #include "symboltable.h"

    entry_t** symbol_table;
    entry_t** constant_table;

    double Evaluate (double lhs_value,int assign_type,double rhs_value);
    int current_dtype;
    int yyerror(char *msg);
%}


%union
{
    double dval;
    entry_t* entry;
    int ival;
    char cval;
}
```

```
%token <entry> IDENTIFIER

 /* Constants */
%token <dval> DEC_CONSTANT HEX_CONSTANT FLOAT_CONSTANT
%token <cval> STRING_CONSTANT
%token STRING

 /* Logical and Relational operators */
%token LOGICAL_AND LOGICAL_OR LS_EQ GR_EQ EQ NOT_EQ

 /* Short hand assignment operators */
%token MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN SUB_ASSIGN
%token LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN XOR_ASSIGN OR_ASSIGN
%token INCREMENT DECREMENT

 /* Data types */
%token SHORT INT FLOAT CHAR LONG LONG_LONG SIGNED UNSIGNED CONST VOID

 /* Keywords */
%token IF FOR WHILE CONTINUE BREAK RETURN

%type <dval> expression
%type <dval> sub_expr
%type <dval> constant
%type <dval> unary_expr
%type <dval> arithmetic_expr
%type <dval> assignment_expr
%type <entry> lhs
%type <ival> assign_op

%start starter

%left ','
%right '='
%left LOGICAL_OR
%left LOGICAL_AND
%left EQ NOT_EQ
%left '<' '>' LS_EQ GR_EQ
%left '+' '-'
%left '*' '/' '%'
%right '!'


%nonassoc UMINUS
%nonassoc LOWER_THAN_ELSE
```

```
%nonassoc ELSE


%%

 /* Program is made up of multiple builder blocks. */
starter: starter builder
           |builder;

 /* Each builder block is either a function or a declaration */
builder: function|
      declaration;

 /* This is how a function looks like */
function: type IDENTIFIER '(' argument_list ')' compound_stmt;

 /* Now we will define a grammar for how types can be specified */

type :data_type pointer
    |data_type;

pointer: '*' pointer
    |'*'
    ;

data_type :sign_specifier type_specifier
    |type_specifier
    ;

sign_specifier :SIGNED
    |UNSIGNED
    ;

type_specifier :INT                 {current_dtype = INT;}
    |FLOAT                        {current_dtype = FLOAT;}
    |CHAR                         {current_dtype = CHAR;}
    |VOID                         {current_dtype = VOID;}
    |SHORT INT                      {current_dtype = SHORT;}
    |SHORT                        {current_dtype = SHORT;}
    |LONG                         {current_dtype = LONG;}
    |LONG INT                       {current_dtype = LONG;}
    |LONG_LONG                        {current_dtype = LONG_LONG;}
    |LONG_LONG INT                      {current_dtype = LONG_LONG;}
    ;

 /* grammar rules for argument list */
 /* argument list can be empty */
```

```
argument_list :arguments
    |
    ;
 /* arguments are comma separated TYPE ID pairs */
arguments :arguments ',' arg
    |arg
    ;

 /* Each arg is a TYPE ID pair */
arg :type IDENTIFIER
   ;

 /* Generic statement. Can be compound or a single statement */
stmt:compound_stmt
    |single_stmt
    ;

 /* The function body is covered in braces and has multiple statements. */
compound_stmt :'{' statements '}'
    ;

statements:statements stmt
    |
    ;

 /* Grammar for what constitutes every individual statement */
single_stmt :if_block
    |for_block
    |while_block
    |declaration
    |function_call ';'
    |RETURN ';'
    |CONTINUE ';'
    |BREAK ';'
    |RETURN sub_expr ';'
    |';'
    ;

for_block:FOR '(' expression_stmt1 expression_stmt2 expression_stmt3 ')' stmt
    ;



if_block:IF '(' expression ')' stmt %prec LOWER_THAN_ELSE
                |IF '(' expression ')' stmt ELSE stmt
    ;
```

```
while_block: WHILE '(' expression   ')' stmt
        ;

declaration:type declaration_list ';'
            |declaration_list ';'
            | unary_expr ';'

declaration_list: declaration_list ',' sub_decl
        |sub_decl;

sub_decl: assignment_expr
    |IDENTIFIER                  {$1 -> data_type = current_dtype;}
    |array_index
    /*|struct_block ';'*/
    ;

/* This is because we can have empty expression statements inside for loops */
/*expression_stmt:expression ';'
    |';'
    ;
*/
expression:
    expression ',' sub_expr                    {$$ = $1,$3;}
    |sub_expr                          {$$ = $1;}
        ;

sub_expr:
    sub_expr '>' sub_expr                 {$$ = ($1 > $3);}
    |sub_expr '<' sub_expr                {$$ = ($1 < $3);}
    |sub_expr EQ sub_expr                 {$$ = ($1 == $3);}
    |sub_expr NOT_EQ sub_expr                {$$ = ($1 != $3);}
    |sub_expr LS_EQ sub_expr                {$$ = ($1 <= $3);}
    |sub_expr GR_EQ sub_expr                {$$ = ($1 >= $3);}
    |sub_expr LOGICAL_AND sub_expr            {$$ = ($1 && $3);}
    |sub_expr LOGICAL_OR sub_expr            {$$ = ($1 || $3);}
    |'!' sub_expr                      {$$ = (!$2);}
    |arithmetic_expr                    {$$ = $1;}
    |assignment_expr                    {$$ = $1;}
    |unary_expr                      {$$ = $1;}
    /* |IDENTIFIER                      {$$ = $1->value;}
    |constant                        {$$ = $1;} */
        //|array_index
    ;


assignment_expr :lhs assign_op arithmetic_expr    {$<dval>$ = $1->value =
Evaluate($1->value,$2,$3);}
```

```
    |lhs assign_op array_index                   {$$ = 0;}
    |lhs assign_op function_call                 {$$ = 0;}
    |lhs assign_op unary_expr                    {$$ = $1->value = Evaluate($1->value,$2,$3);}
    |unary_expr assign_op unary_expr             {$$ = 0;}
    ;

expression_stmt3: assignment_expr
    | unary_expr
    |
    ;

unary_expr: lhs INCREMENT                        {$$ = $1->value = ($1->value)++;}
    |lhs DECREMENT                               {$$ = $1->value = ($1->value)--;}
    |DECREMENT lhs                               {$$ = $2->value = --($2->value);}
    |INCREMENT lhs                               {$$ = $2->value = ++($2->value);}

lhs:IDENTIFIER                                   {$$ = $1; if(! $1->data_type) $1->data_type =
current_dtype;}
    //|array_index
    ;

assign_op:'='                                    {$$ = '=';}
    |ADD_ASSIGN                                  {$$ = ADD_ASSIGN;}
    |SUB_ASSIGN                                  {$$ = SUB_ASSIGN;}
    |MUL_ASSIGN                                  {$$ = MUL_ASSIGN;}
    |DIV_ASSIGN                                  {$$ = DIV_ASSIGN;}
    |MOD_ASSIGN                                  {$$ = MOD_ASSIGN;}
    ;

arithmetic_expr: arithmetic_expr '+' arithmetic_expr    {$$ = $1 + $3;}
    |arithmetic_expr '-' arithmetic_expr         {$$ = $1 - $3;}
    |arithmetic_expr '*' arithmetic_expr         {$$ = $1 * $3;}
    |arithmetic_expr '/' arithmetic_expr         {$$ = ($3 == 0) ? yyerror("Divide by 0!") :
($1 / $3);}
    |arithmetic_expr '%' arithmetic_expr         {$$ = (int)$1 % (int)$3;}
    |'(' arithmetic_expr ')'                     {$$ = $2;}
    |'-' arithmetic_expr %prec UMINUS            {$$ = -$2;}
    |IDENTIFIER                                  {$$ = $1 -> value;}
    |constant                                    {$$ = $1;}
    ;

constant: DEC_CONSTANT                           {$$ = $1;}
    |HEX_CONSTANT                                {$$ = $1;}
    |FLOAT_CONSTANT                              {$<dval>$ = $<dval>1;}
    |STRING_CONSTANT                             {$<cval>$ = $<cval>1;}
    ;
```

```
expression_stmt1: assignment_expr ';'
    | ';'
;

expression_stmt2: sub_expr ';'
|';'
;

array_index: IDENTIFIER '[' sub_expr ']'

function_call: IDENTIFIER '(' parameter_list ')'
             |IDENTIFIER '(' ')'
             ;

parameter_list:
              parameter_list ','  parameter
              |parameter
              ;

parameter: sub_expr
                 |STRING

        ;
%%

#include "lex.yy.c"
#include <ctype.h>


double Evaluate (double lhs_value,int assign_type,double rhs_value)
{
    switch(assign_type)
    {
        case '=': return rhs_value;
        case ADD_ASSIGN: return (lhs_value + rhs_value);
        case SUB_ASSIGN: return (lhs_value - rhs_value);
        case MUL_ASSIGN: return (lhs_value * rhs_value);
        case DIV_ASSIGN: return (lhs_value / rhs_value);
        case MOD_ASSIGN: return ((int)lhs_value % (int)rhs_value);
    }
}

int main(int argc, char *argv[])
{
    symbol_table = create_table();
    constant_table = create_table();
    yyin = fopen("test-case-1.c", "r");

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


    printf("\n\tSymbol table");
    display(symbol_table);
    printf("\n\Constant table");
    display(constant_table);


    fclose(yyin);
    return 0;
}

int yyerror(char *msg)
{
    printf("Line no: %d Error message: %s Token: %s\n", yylineno, msg, yytext);
}
```

Features

- Nested code blocks
- +=, -= supported
- for loops and nested for loops
- integer assignment and declaration
- if-else handled
- array assignment, array declaration
- comma separated initialization/declaration e.g int a=10,b;
- support for short, long , long long , signed and unsigned
- while statements supported
- unary expressions supported in both lhs, rhs and standalone statements
- display specific errors with line numbers

3. Test Cases

Without Errors:

S.No	Test Case	Expected Output	Status

1	int main(){
	 int a=100;
	}
		Parsing complete
		
		Symbol table
		===
		lexeme value data-type
		===
		a 100 283
		
		===
		
		Constant table
		===
		lexeme value data-type
		===
		100 259 0
		===
			PASS

2	int main(){
	 float a[10];
	}
		Parsing complete
		
		Symbol table
		===
		lexeme value data-type
		===
		a 10 283
		===
		
		Constant table
		===
		lexeme value data-type
		===
		10 259 0
		===
			PASS

3	int main(){
	unsigned int a = 0x0f;
	}
		Parsing complete
		
		Symbol table
		===
		lexeme value data-type
		===
		a 0x0f 283
		===
		
		Constant table
		===
			PASS

		```
		lexeme          value           data-type
		=============================================
		0x0f            259             0
		=============================================
		```

4	int main(){
	total = x + y;
	}
		```
		Parsing complete

		        Symbol table
		=============================================
		lexeme          value           data-type
		=============================================
		total           2147483647          283
		=============================================

		Constant table
		=============================================
		lexeme          value           data-type
		=============================================

		=============================================
		```
			PASS

5	Int main(){
	printf("This is a string");
	}
		```
		Parsing complete

		        Symbol table
		=============================================
		lexeme          value           data-type
		=============================================
		printf          2147483647        0
		=============================================

		Constant table
		=============================================
		lexeme          value           data-type
		=============================================
		"This is a string"      2147483647      0
		=============================================
		```
			PASS

6	Int main(){
	Str = “ \” hello \” ”
	}
		```
		Parsing complete

		        Symbol table
		=============================================
		lexeme          value           data-type
		=============================================
		Str           2147483647        0
		=============================================

		Constant table
		=============================================
		lexeme          value           data-type
		=============================================
		```
			PASS

		```
		“ \” hello \” ”          2147483647          0
		==============================================
		```

With Errors:

S.No	Test Case	Expected Output	Status

1	int main(){
	 /* To make things /* nested multi-line comment */ interval */
	}
		```
		Parsing failed

		        Symbol table
		==============================================
		 lexeme          value          data-type
		==============================================

		==============================================

		Constant table
		==============================================
		 lexeme          value          data-type
		==============================================

		==============================================
		```
			PASS

2	int main(){
	str= “star
	}
		```
		Parsing failed

		        Symbol table
		==============================================
		 lexeme          value          data-type
		==============================================
		  str          2147483647      0
		==============================================

		Constant table
		==============================================
		 lexeme          value          data-type
		==============================================

		==============================================
		```
			PASS

3	#inc
	int main(){}
		```
		Parsing failed

		        Symbol table
		==============================================
		 lexeme          value          data-type
		==============================================

		==============================================

		Constant table
		```
			PASS

		== lexeme value data-type == ==	

4	int main(){
	10 = a;
	}
		Parsing failed
		
		Symbol table
		==
		lexeme value data-type
		==
		a 2147483647 283
		==
		
		Constant table
		==
		lexeme value data-type
		==
		10 259 0
		==
			PASS

5	int main(){
	9f = d @ j ;
	}
		Parsing failed
		
		Symbol table
		==
		lexeme value data-type
		==
		printf 2147483647 0
		i 2147483647 0
		main 2147483647 0
		func 2147483647 0
		==
		
		Constant table
		==
		lexeme value data-type
		==
		"%d\n" 2147483647 0
		==
			PASS

6	int main(){
	int atgd$;
	}
		Parsing failed
		
		Symbol table
		==
		lexeme value data-type
		==
		atgd 2147483647 283
		==
			PASS

		Constant table == lexeme value data-type == ==	

Test Case 1:

```
int main(){

  int a = 10;
  int b = 10

  for(a = 2 a<3; a++)
  b = b + 1;

  printf (" This string is enclosed in double quotes ");
  printf (" This string is not enclosed in double quotes );
  return 0;
}
```

Output 1:

```
Line no: 6 Error message: syntax error Token: for

Parsing failed

         Symbol table
=====================================================
 lexeme          value          data-type
=====================================================
 a             10             283
 main            2147483647        0
 b                  10                  283
====================================================


Constant table
====================================================
 lexeme             value               data-type
====================================================
 10                 259                 0
====================================================
```

Test Case 2:

```
int main()
{
        int array1[10];
        int array2[20;

        for(int a = 3; a<4; a++
        {
                a = array1[0];
        }

        func();
}

int func()
{
        return 20;
```

Output 2 :

```
Line no: 4 Error message: syntax error Token: ;

Parsing failed

          Symbol table
=====================================================
 lexeme              value               data-type
=====================================================
 main              2147483647        0
 array2            2147483647        0
 array1            2147483647        0
=====================================================

Constant table
=====================================================
 lexeme              value               data-type
=====================================================
 10                259               0
 20                259               0
=====================================================
```

Test Case 3:

```
int main()
{

  `

 @ -
 total = x @ y;

          int a = 4;
```

```
        if(a > 0 )
        {
                printf("a is positive");
                a * 2 = a;
        }
        else
        {
                printf("a is negative");
                a = a * 2;
        }

}
```

Output 3 :

```
Line no: 4 Error message: syntax error Token: `
Parsing failed

        Symbol table
=====================================================
 lexeme          value           data-type
=====================================================
 main           2147483647        0
=====================================================

Constant table
=====================================================
 lexeme          value           data-type
=====================================================
=====================================================
```

Test Case 4:

```
int main()
{
  int x, y;
  long long int total, diff;
  int *ptr;
  int a = 86;

  10 = a;
}
```

Output 4 :

```
Line no: 8 Error message: syntax error Token: 10


Parsing failed


        Symbol table
=====================================================
 lexeme          value            data-type
=====================================================
 total          2147483647         287
 x             2147483647          283
 a             86                283
 main            2147483647          0
 y             2147483647          283
 ptr            2147483647          283
 diff           2147483647          287
=====================================================


Constant table
=====================================================
 lexeme          value          data-type
==========================================================
 86             259           0
 10             259           0
==========================================================
```

Test Case 5:

```
int main()
{
  unsigned int a = 0x0f;
  long int mylong = 123456l;
  long int i, j;

  for(i=0; i < 10; i++)
  {
    for(j=10; j > 0; j--)
             {
                          printf("%d",i);
    }
  }

  diff = x - y;
  int rem = x % y;
  printf ("Total = %d \n", total);

  int result = IncreaseBy10(x);
}

int IncreaseBy10(int x)
{
             return x + 10;
}
```

Output 5 :

```
    Parsing complete

        Symbol table
==================================================
 lexeme          value          data-type
==================================================
 total          2147483647       0
 result          2147483647       283
 printf          2147483647       0
 j              10              286
 x              2147483647       0
 i              0              286
 a              15              283
 mylong          123456           286
 main            2147483647       0
 rem             0              283
 y              2147483647       0
 diff            0              286
 IncreaseBy10       2147483647       0
==================================================


Constant table
==================================================
 lexeme          value          data-type
==================================================
 10             259             0
 123456l          259             0
 "%d"            2147483647       0
 0x0f            260             0
```

```
"Total = %d \n"      2147483647          0
0                    259                 0
=========================================================
```

Test Case 6:

```
int main()
{
  int c = 0,d,e,f;

  c = c + 1;  // c = 1
  d = c * 5;  // d = 5
  e = d / 4;  // e = 1
  f = d % 3;  // f = 2
  f = f / 0;
}
```

Output 6 :

```
Parsing complete


          Symbol table
=========================================================
 lexeme             value              data-type
=========================================================
 c                  1                  283
 main                  2147483647          0
 d                  5                  283
 f                  2                  283
 e                  1.250000           283
=========================================================
Constant table
=======================================================
 lexeme          value           data-type
=======================================================
 5             259             0
 3             259             0
 1             259             0
 0             259             0
 4             259             0
=======================================================
```

4. Implementation

The lexer code submitted in the previous phase took care of most of the features of C using regular expressions. Some special corner cases were taken care using custom reg-ex. These were:

- The Regex for Identifiers
- Multiline comments should be supported
- Literals
- Error Handling for Incomplete String
- Error Handling for Nested Comments

The parser code requires exhaustive token recognition and because of this reason, we utilised the lexer code given under the C specifications with the parser. The parser implements C grammar using a number of production rules.

Parser takes tokens from the lexer output, one at a time and applies the corresponding production rules to generate the symbol table with the variable and function types. If the parsing is not successful, the parser outputs the line number with the corresponding error. We have written a function, insert(), which takes a string (name of the identifier) as input and –

1. If the identifier is already present in the symbol table, does not do any further steps.

2. If not already present, creates a new struct entry, adds it to the symbol table, and returns the struct

5.Results and Future Works:

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.

Future Works:

The yacc script presented in this report takes care of all the parsing 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 function parameters and other corner cases, and thus make it more effective.

6.References

1. Lex and Yacc By John R. Levine, Tony Mason, Doug Brown

2. Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

3. https://en.wikipedia.org/wiki/Symbol_table

4.https://www.cse.iitb.ac.in/~br/courses/cs699-autumn2013/refs/lextut-victor-eijkhout.pdf

5.https://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/tutorial-large.pdf

6.http://dinosaur.compilertools.net/lex/

7.Jenkins Hash Function on Wikipedia :

https://en.wikipedia.org/wiki/Jenkins_hash_function#one-at-a-time