

Parser for the C Language



National Institute of Technology Karnataka, Surathkal

Date: 25th February 2019

Submitted To: Dr. P. Santhi Thilagam

Dept. of Computer Science, NITK Surathkal

Group Members:

1. Adarsh Kumar (171CO204)
2. Saurabh Singhal (171CO240)
3. Rahul Kumar (17CO232)

Abstract:

This report contains the details of the tasks finished as a part of 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. In the previous submission, we were using the lexical analyzer to generate the stream of tokens from the source code. We used look-ahead for checking errors in comments and some other lexical errors. But lexical analyzer cannot detect errors in the structure of a language (syntax), unbalanced parenthesis, etc. These errors are handled by a parser. After the lexical phase, the compiler enters the syntax analysis phase. This analysis is done by a parser. The parser uses the stream of tokens from the scanner and assigns them datatype if they are identifiers. The parser code has the functionality of taking input through a file or through standard input. This makes it more user-friendly and efficient at the same time.

Contents:

Page No

1. Introduction	3
a. Parser/Syntactic Analysis	3
b. Yacc Script	3
c. C Program	4
2. Design of Programs	5
a. Code	5
b. First of Grammar	22
c. Follow of Grammar	23
d. Explanation	24
3. Test Cases	
a. Without Errors	25
b. With Errors	26
4. Implementation	28
5. Results	29
6. Future work	31
7. References	31

Introduction:

Syntactic Analysis and Parser:

After the lexical analysis stage, we get the stream of tokens from source C code which is given as input to the parser. Parser verifies that a string of token names can be generated by the grammar of the source language. We expect the parser to report any syntax errors in an intelligible manner and to recover from the commonly occurring errors to continue processing the remainder of the program. Parser detects the following types of errors:

1. Errors in structure
2. Missing operator
3. Misspelled keywords
4. Unbalanced parenthesis

Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

There are generally 3 types of parsers for grammars:

1. Universal
2. Top-down
3. Bottom-up

The methods commonly used in compilers can be classified as being either top-down (parse from root to leaves) or bottom-up (parse from leaves to root).

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 lexer can be used to make a simple parser. But it needs making extensive use of the user-defined states.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input Parser for C Language Page 5 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 specifications.

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.

In the rules section, each grammar rule defines a symbol in terms of:

1. Other symbols
2. Tokens (or terminal symbols) which come from the lexer.

Each rule can have an associated action, which is executed after all the component symbols of the rule have been parsed. Actions are basically C-program statements surrounded by curly braces.

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 c_parser.y
```

2. Compile the flex script using Flex tool

```
$ flex c_lexer.l
```

3. After compiling the lex file, a 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 -o compiler lex.yy.c y.tab.h y.tab.c -ll -ly
```

5. 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:

Lexer Code:

```
%{  
  
    #include <stdio.h>  
    #include <string.h>  
    #include "y.tab.h"  
  
    #define ANSI_COLOR_RED          "\x1b[31m"  
    #define ANSI_COLOR_GREEN        "\x1b[32m"  
    #define ANSI_COLOR_YELLOW       "\x1b[33m"  
    #define ANSI_COLOR_BLUE         "\x1b[34m"  
    #define ANSI_COLOR_MAGENTA      "\x1b[35m"  
    #define ANSI_COLOR_CYAN         "\x1b[36m"  
    #define ANSI_COLOR_RESET        "\x1b[0m"  
  
    struct symboltable  
    {  
        char name[100];  
        char class[100];  
        char type[100];  
        char value[100];  
        int lineno;  
        int length;  
    }ST[1001];  
  
    struct constanttable  
    {  
        char name[100];  
        char type[100];  
        int length;  
    }CT[1001];  
  
    int hash(char *str)  
    {  
        int value = 0;  
        for(int i = 0 ; i < strlen(str) ; i++)  
        {  
            value = 10*value + (str[i] - 'A');  
            value = value % 1001;  
            while(value < 0)  
                value = value + 1001;  
        }  
    }
```

```

        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)%1001)
            {
                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)%1001)
            {
                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);
            insertSTline(str1, yylineno);
            return;
        }

        int pos = 0;

        for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
        {
            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 < 1001 ; i++)
    {
        if(strcmp(ST[i].name, str1)==0)
        {
            strcpy(ST[i].type, str2);
        }
    }
}

```



```

    }
}

void insertSTvalue(char *str1, char *str2)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name,str1)==0)
        {
            strcpy(ST[i].value,str2);
        }
    }
}

void insertSTline(char *str1, int line)
{
    for(int i = 0 ; i < 1001 ; 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);
            return;
        }

        int pos = 0;

        for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
        {
            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 | %10s | %10s\n", "SYMBOL",
"CLASS", "TYPE", "VALUE", "LINE NO");
    for(int i=0; i<81; i++) {
        printf("-");
    }
    printf("\n");
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(ST[i].length == 0)
        {
            continue;
        }
        printf("%10s | %15s | %10s | %10s | %10d\n", ST[i].name, ST[i].class, ST[i].type, ST[i].value, ST[i].lineno);
    }
}

void printCT()
{
    printf("%10s | %15s\n", "NAME", "TYPE");
    for(int i=0; i<81; i++) {
        printf("-");
    }
    printf("\n");
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(CT[i].length == 0)
            continue;

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

```

```

    }
}
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(','); }
("{")    { return('{'); }
("}")    { return('}'); }
("("     { return('('); }
(")")    { return(')'); }
("[ "| "<:") { return('['); }
("]" | ">:") { return(']'); }
":"      { return(':'); }
"."      { return('.'); }

"char"      { strcpy(curtype,yytext); insertST(yytext,
"Keyword"); return CHAR;}
"double"    { strcpy(curtype,yytext); insertST(yytext, "Keyword");
return DOUBLE;}
"else"      { insertSTline(yytext, yylineno);
insertST(yytext, "Keyword"); return ELSE;}
"float"     { strcpy(curtype,yytext); insertST(yytext,
"Keyword");return FLOAT;}
"while"     { insertST(yytext, "Keyword"); return WHILE;}
"do"        { insertST(yytext, "Keyword"); return DO;}
"for"       { insertST(yytext, "Keyword"); return FOR;}
"if"        { insertST(yytext, "Keyword"); return IF;}

```

```

"int"          { strcpy(curtype,yytext);      insertST(yytext,
"Keyword");return INT;}
"long"         { strcpy(curtype,yytext);      insertST(yytext,
"Keyword"); return LONG;}
"return"       { insertST(yytext, "Keyword"); return RETURN;}
"short"        { strcpy(curtype,yytext);      insertST(yytext,
"Keyword"); return SHORT;}
"signed"       { strcpy(curtype,yytext); insertST(yytext, "Keyword");
return SIGNED;}
"sizeof"       { insertST(yytext, "Keyword"); return SIZEOF;}
"struct"       { strcpy(curtype,yytext); insertST(yytext, "Keyword");
return STRUCT;}
"unsigned"     { insertST(yytext, "Keyword"); return UNSIGNED;}
"void"         { strcpy(curtype,yytext);      insertST(yytext,
"Keyword"); return VOID;}
"break"        { insertST(yytext, "Keyword"); return BREAK;}


"++"           { return increment_operator; }
"--"           { return decrement_operator; }
"<<"          { return leftshift_operator; }
">>"          { return rightshift_operator; }
"<="          { return lessthan_assignment_operator; }
"<"           { return lessthan_operator; }
">="          { return greaterthan_assignment_operator; }
">"           { return greaterthan_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 leftshift_assignment_operator; }
">>="         { return rightshift_assignment_operator; }
"&="          { return AND_assignment_operator; }
"^="          { return XOR_assignment_operator; }
"|="          { return OR_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-zA-Z]([a-zA-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;
    }

%%

```

Parser Code:

```

%{
    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 ANSI_COLOR_RED        "\x1b[31m"
    #define ANSI_COLOR_GREEN      "\x1b[32m"
    #define ANSI_COLOR_CYAN       "\x1b[36m"
    #define ANSI_COLOR_RESET      "\x1b[0m"

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

%}

%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
%token RETURN MAIN
%token VOID
%token WHILE FOR DO
%token BREAK
%token ENDIF
%expect 2

%token identifier
%token          integer_constant      string_constant      float_constant
character_constant

%nonassoc ELSE

```

```

%right leftshift_assignment_operator rightshift_assignment_operator
%right XOR_assignment_operator OR_assignment_operator
%right AND_assignment_operator 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      lessthan_assignment_operator      lessthan_operator
greaterthan_assignment_operator greaterthan_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 program

%%
program
    : declaration_list;

declaration_list
    : declaration D

D
    : declaration_list
    | ;

declaration
    : variable_declaration
    | function_declaration
    | structure_definition;

variable_declaration
    : type_specifier variable_declaration_list ';'
    | structure_declaration;

```

```

variable_declaration_list
    : variable_declaration_identifier V;

V
    : ',' variable_declaration_list
    | ;

variable_declaration_identifier
    : identifier { ins(); } vdi;

vdi : identifier_array_type | assignment_operator expression ;

identifier_array_type
    : '[' initialization_params
    | ;

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

initialization
    : string_initialization
    | array_initialization
    | ;

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

unsigned_grammar
    : INT | LONG long_grammar | SHORT short_grammar | ;

signed_grammar
    : INT | LONG long_grammar | SHORT short_grammar | ;

long_grammar
    : INT | ;

short_grammar
    : INT | ;

```



```

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

V1 : variable_declaration V1 | ;

structure_declaration
    : STRUCT identifier variable_declaration_list;

function_declaration
    :                                     function_declaration_type
function_declaration_param_statement;

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

function_declaration_param_statement
    : params ')' statement;

params
    : parameters_list | ;

parameters_list
    : type_specifier parameters_identifier_list;

parameters_identifier_list
    : param_identifier parameters_identifier_list_breakup;

parameters_identifier_list_breakup
    : ',' parameters_list
    | ;

param_identifier
    : identifier { ins(); } param_identifier_breakup;

param_identifier_breakup
    : '[' ']'
    | ;

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

```

```

compound_statement
    : '{' statment_list '}' ;

statment_list
    : statement statment_list
    | ;

expression_statment
    : expression ';'
    | ';' ;

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

conditional_statements_breakup
    : ELSE statement
    | ;

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

return_statement
    : RETURN return_statement_breakup;

return_statement_breakup
    : ';'
    | expression ';' ;

break_statement
    : BREAK ';' ;

string_initilization
    : assignment_operator string_constant { insV(); };

array_initialization
    : assignment_operator '{' array_int_declarations '}';

array_int_declarations
    : integer_constant array_int_declarations_breakup;

```

```

array_int_declarations_breakup
    : ',' array_int_declarations
    | ;

expression
    : mutable expression_breakup
    | simple_expression ;

expression_breakup
    : 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_breakup;

simple_expression_breakup
    : OR_operator and_expression simple_expression_breakup
    | ;

and_expression
    : unary_relation_expression and_expression_breakup;

and_expression_breakup
    : AND_operator unary_relation_expression
and_expression_breakup
    | ;

unary_relation_expression
    : exclamation_operator unary_relation_expression
    | regular_expression ;

regular_expression
    : sum_expression regular_expression_breakup;

regular_expression_breakup
    : relational_operators sum_expression
    | ;

relational_operators

```

```

        :      greaterthan_assignment_operator      |
lessthan_assignment_operator | greaterthan_operator
        |      lessthan_operator      |      equality_operator      |
inequality_operator ;

sum_expression
        : sum_expression sum_operators term
        | term ;

sum_operators
        : add_operator
        | subtract_operator ;

term
        : term MULOP factor
        | factor ;

MULOP
        :      multiplication_operator      |      division_operator      |
modulo_operator ;

factor
        : immutable | mutable ;

mutable
        : identifier
        | mutable mutable_breakup;

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

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

call
        : identifier '(' arguments ')';

arguments
        : arguments_list | ;

arguments_list
        : expression A;

```

```

A
    : ',' expression A
    | ;

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

%%

extern FILE *yyin;
extern int yylineno;
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(ANSI_COLOR_GREEN "Status: Parsing Complete - Valid"
ANSI_COLOR_RESET "\n");
        printf("%30s" ANSI_COLOR_CYAN "SYMBOL" "TABLE"
ANSI_COLOR_RESET "\n", " ");
        printf("%30s %s\n", " ", "-----");
        printST();

        printf("\n\n%30s" ANSI_COLOR_CYAN "CONSTANT" "TABLE"
ANSI_COLOR_RESET "\n", " ");
        printf("%30s %s\n", " ", "-----");
        printCT();
    }
}

void yyerror(char *s)
{
    printf("%d %s %s\n", yylineno, s, yytext);
}

```

```
        flag=1;
        printf(ANSI_COLOR_RED "Status: Parsing Failed - Invalid\n"
ANSI_COLOR_RESET);
    }

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

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

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

First Set

Non-Terminal Symbol	First Set
Include	Include
Array_Notation	Array_Notation
Type	Type
IDENTIFIER	IDENTIFIER
'{'	'{'
Formal_Param_List	Formal_Param_List
'}'	'}'
Compound_Statement	Compound_Statement
{	{
{symbol_table_insert(symbol_table,\$2,datatype,yylineno);trace("Formal_Param_List	{symbol_table_insert(symbol_table,\$2,datatype,yylineno);trace("Formal_Param_List
Rule	Rule
1\n");}	1\n");}
INT	INT
{strcpy(datatype,	{strcpy(datatype,
\$1);}	\$1);}
SHORT	SHORT
LONG	LONG
UNSIGNED	UNSIGNED
SIGNED	SIGNED
'T'	'T'
'J'	'J'
Assignment_Operator	Assignment_Operator
Expression	Expression
{symbol_table_insert(symbol_table,\$1,datatype,yylineno);trace("Define_Assign	{symbol_table_insert(symbol_table,\$1,datatype,yylineno);trace("Define_Assign
{trace("Assignment	{trace("Assignment
'='	'='
Logical_Expression	Logical_Expression
Relational_Expression	Relational_Expression
Additive_Expression	Additive_Expression
Multiplicative_Expression	Multiplicative_Expression
Primary	Primary
'{'	'{'
Statement_List	Statement_List
'}'	'}'
Statement	Statement
While_Statement	While_Statement
RETURN	RETURN
','	','
{trace("Return_Statement	{trace("Return_Statement
Call\n");}	Call\n");}
WHILE	WHILE
DO	DO
FOR	FOR
Assignment	Assignment
IF	IF
Else_Statement	Else_Statement
ELSE	ELSE
Param_List	Param_List
{symbol_table_insert(symbol_table,	{symbol_table_insert(symbol_table,
\$1,	\$1,
"Function",	"Function",
yylineno);trace("Function	yylineno);trace("Function
'#'	'#'
INCLUDE	INCLUDE
LESSER_THAN	LESSER_THAN
GREATER_THAN	GREATER_THAN
Include_Statement	Include_Statement
Identifier_List	Identifier_List
:	Include, Array_Notation, Type, INT, SHORT, LONG, UNSIGNED, SIGNED, IDENTIFIER, Expression, '=', Logical_Expression, Relational_Expression, Additive_Expression, Multiplicative_Expression, Primary, '{', '}', Statement, While_Statement, RETURN, WHILE, DO, FOR, IF, ELSE, '#', Include_Statement, Type

Follow Set

Non-Terminal Symbol Follow Set

\$
:

Predict Set

#	Expression	Predict
1 → Include		Include
2 → Array_Notation		Array_Notation
3 → Type IDENTIFIER '(' Formal_Param_List ')' Compound_Statement { → Type IDENTIFIER		Type
4 {symbol_table_insert(symbol_table,\$2,datatype,yylineno);trace("Formal_Param_List Type Rule 1\n");}		
5 → INT {strcpy(datatype, \$1);}		INT
6 → SHORT		SHORT
7 → LONG		LONG
8 → UNSIGNED		UNSIGNED
9 → SIGNED		SIGNED
10 → IDENTIFIER '[' ']' { → IDENTIFIER Assignment_Operator Expression		IDENTIFIER
11 {symbol_table_insert(symbol_table,\$1,datatype,yylineno);trace("Define_Assign Rule 1\n");}		IDENTIFIER
12 → Expression		Expression
13 → IDENTIFIER Assignment_Operator Expression {trace("Assignment Rule 1\n");}		IDENTIFIER
14 → '='		'='
15 → Logical_Expression		Logical_Expression
16 → Relational_Expression		Relational_Expression
17 → Additive_Expression		Additive_Expression
18 → Multiplicative_Expression		Multiplicative_Expression
19 → Primary		Primary
20 → '(' Expression ')'		'('
21 → '{' Statement_List '}'		'{'
22 → Statement Statement_List		Statement
23 → While_Statement		While_Statement
24 → RETURN Expression ';' {trace("Return_Statement Call\n");}		RETURN
25 → WHILE '(' Expression ')' Statement		WHILE
26 → DO Compound_Statement WHILE '(' Expression ')' ';'		DO
27 → FOR '(' Assignment ';' Expression ';' Assignment ')' Statement		FOR
28 → IF '(' Expression ')' Statement Else_Statement		IF
29 → ELSE Compound_Statement		ELSE
30 → IDENTIFIER '(' Param_List ')' {symbol_table_insert(symbol_table, \$1, "Function", yylineno);trace("Function Call\n");}		IDENTIFIER
31 → '# INCLUDE LESSER_THAN IDENTIFIER GREATER_THAN		'#'
32 → Include_Statement		Include_Statement
33 : → Type Identifier_List ';'		Type

Explanation:

The lex code is detecting the tokens from the source code and returning the corresponding token to the parser. In phase 1 we were just printing the token and now we are returning the token so that parser uses it for further computation. We are using the symbol table and constant table of the previous phase only. We added functions like **insertSTtype()**, **insertSTvalue()** and **insertSTline()** to the existing functions. Lexical Analyser installs the token in the symbol table whereas parser calls these functions to add the value of attributes like data type, value assigned to identifier and where the identifier was declared i.e. updates the information in the symbol table.

Declaration Section

In this section we have included all the necessary header files,function declaration and flag that was needed in the code.

Between 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.This ensures the grammar we are giving to the parser is unambiguous as LALR(1) parser cannot work with ambiguous grammar.

Rules Section

In this section production rules for entire C language is written. The rules are written in such a way that there is no left recursion and the grammar is also deterministic. Non-deterministic grammar was converted to deterministic by applying left factoring. This was done so that grammar is for LL(1) parser. This is so because all LL(1) grammar are LALR(1) according to the concepts. The grammar productions do the syntax analysis of the source code. When a complete statement with proper syntax is matched by the parser.

C-Program Section

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

Test Cases:

Without Errors:

Test Case 1:

```
#include<stdio.h>
/*Test case for operator,nested loops, delimiters, function,
assignments;*/
int fun(int x){
return x*x;
}
void main(){
int a=2,b,d,e,f,g,h,i,n=10,x;
d=a*b;
e=a/b;
f=a%b;
g=a&&b;
h=a*a+b*b;
h=fun(b);
for (i=0;i<n;i++){
if(i<10){
int x;
while(x<10){
x++;
}
}
}
}
```

Test Case 2:

```
#include<stdio.h>
//This test case contains nested conditional statement, array and string
declaration
int main(){
char s[10]="Welcome!!";
char s[]="Welcome!!";
int a[2] = {1, 2};
char S[20];
if(s[0]=='W'){
if(s[1]=='e'){
if(s[2]=='l'){
printf("Welcome!!");
}
else printf("Bug1\n");
}
```

```

}
else printf("Bug2\n");
}
else printf("Bug3\n");

}

```

Test Case 3:

```

#include<stdio.h>
// This test case contains struct and functions
int square(int a){
return(a*a);
}
struct abc{
int a;
char b;
};
int main(){
struct abc A;

```

```

A.a = 2;
int num = 2;
int num2 = square(num);
return 0;
}

```

With Errors:

Test Case 1:

```

// Invalid syntax missing semicolon.
#include<stdio.h>
int main() {
char hello
hello = 'c';

```

Test Case 2:

```

// Error in structure of assignment
#include <stdio.h>
int main() {

```

```
char hello;  
'c' = 'x';  
}
```

Test Case 3:

```
//Unmatched paranthesis  
#include<stdio.h>  
int main(){  
int a = 1, b=0;  
if(a >= 1 && a <= 10)  
b++;  
else{  
b--;  
} }  
}
```

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 of using custom regex. 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.

The parser takes tokens from the lexer output, one at a time and applies the corresponding production rules to append to the symbol table with type, value and line of declaration. If the parsing is not successful, the parser outputs the line number with the corresponding error.

The following functions were written in order to maintain symbol table:

1. **LookupST()** - This function checks whether the token is already present in the symbol table or not. If yes it returns 1 else 0.(Called by Scanner)
2. **InsertST()** - This function installs the token in the symbol table if it is not already present along with the token class.(Called by Scanner)
3. **InsertSTtype()** - This function appends the datatype of the identifier in the symbol table. (Called by Parser).

4. **InsertSTvalue()**- This function appends the value of the identifier in the symbol table. (Called by Parser).
5. **InsertSTline()**- This function appends the line of declaration of the identifier in the symbol table. (Called by Parser).
6. **LookupCT()** - This function checks whether the token is already present in the constant table or not. If yes it returns 1 else 0.(Called by Scanner).
7. **InsertST()** - This function installs the token in the constant table if it is not already present along with the token class.(Called by Scanner)
8. **PrintST()** - This function displays the entire content of the symbol table.
9. **PrintCT()** - This function displays the entire content of the constant table.

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, values and line of declaration. The parser generates error messages in case of any syntactical errors in the test program.

Test Case 1: Operator, Nested loops, Delimiters, Function, Assignments Output:

```

===== Running Testcase 2 =====
Status: Parsing Complete - Valid

SYMBOL TABLE
-----
SYMBOL | CLASS | TYPE | VALUE | LINE NO
-----
a | Identifier | int | 2 | 8
b | Identifier | int | 8 | 2
c | Identifier | int | 8 | 16
d | Identifier | int | 8 | 17
e | Identifier | int | 8 | 9
f | Identifier | int | 8 | 10
g | Identifier | int | 8 | 11
h | Identifier | int | 8 | 12
x | Identifier | char | 3 | 13
char | Keyword |  | 3 | 17
fun | Identifier | int | 3 | 10
return | Keyword |  | 4 | 18
int | Keyword |  | 3 | 10
main | Identifier | void | 7 | 18
void | Keyword |  | 7 | 17

CONSTANT TABLE
-----
NAME | TYPE | VALUE | LINE NO
-----
2 | Number Constant | 2 | 7

```

Output:

```

===== Running TestCase 7 =====
Status: Parsing Complete - Valid
SYMBOL TABLE
=====
SYMBOL | CLASS | TYPE | VALUE | LINE NO
-----
A | Identifier | struct | 0 | 16
a | Identifier | int | 2 | 3
struct | Keyword | Identifier | struct | 0
b | Identifier | char | 0 | 15
num | Identifier | int | 2 | 18
square | Identifier | int | 3 | 14
char | Keyword | Identifier | char | 11
return | Keyword | Identifier | int | 5
int | Keyword | Identifier | int | 3
abc | Identifier | struct | 8 | 8
num2 | Identifier | int | 0 | 19
main | Identifier | int | 14 | 14
printf | Identifier | Identifier | "Square of %d is %d" | 21
num2 | Identifier | Identifier | 17
main | Identifier | Identifier | 12
=====
CONSTANT TABLE
=====
NAME | TYPE
-----
"Square of %d is %d" | String Constant
0 | Number Constant
2 | Number Constant

```

Status : PASS

Test Case 3: Convoluted Conditional Statement, Array and String declaration

Output:

```

===== Running TestCase 11 =====
Status: Parsing Complete - Valid
Parser for C Language
SYMBOL TABLE
=====
SYMBOL | CLASS | TYPE | VALUE | LINE NO
-----
a | Identifier | int | 1
b | Array Identifier | int | 2
c | Identifier | int | 3
d | Array Identifier | char | "kjbk" | 4
e | Array Identifier | char | "jljlj" | 5
f | Array Identifier | int | 3 | 6
g | Identifier | int | 7
h | Identifier | int | 7
i | Identifier | int | 7
char | Keyword | Identifier | int | 4
int | Keyword | Identifier | char | 1
myfunc | Identifier | int | 8
=====
CONSTANT TABLE
=====
NAME | TYPE
-----
"jljlj" | String Constant
100 | Number Constant
"kjbk" | String Constant
10 | Number Constant
0 | Number Constant
3 | Number Constant

```

Status : PASS

Invalid Test Cases

Test Case 1: Invalid syntax missing semicolon

Output:

```
===== Running TestCase 14 =====  
ERROR at line no. 6  
@  
6 syntax error @  
Status: Parsing Failed - Invalid
```

Status : PASS

Test Case 2: Error in struct of assignment

Output:

```
===== Running TestCase 15 =====  
ERROR at line no. 4  
9 Status: PASS  
4 syntax error 9  
Status: Parsing Failed - Invalid
```

Status : PASS

Test Case 3: Unmatched parentheses

Output:

```
Test Case 3: Unmatched parentheses  
===== Running TestCase 16 =====  
Error in Pre-Processor directive at line no. 1  
#  
1 syntax error # ERROR at line no. 5  
Status: Parsing Failed - Invalid
```

Status : PASS

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:

1. <http://dinosaur.compilertools.net/>
2. <http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf>
3. Compilers: Principles, Techniques, and Tools: Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman