Phase - 2

Syntax Analysis (C Language)

<u>Team Members</u> –

1. Bharadwaja M Chittapragada (211CS216)

2.Rishi Diwaker (211CS243)

3.Rohit Kumar (211CS244)

Abstract

This report outlines the progress made during Phase Two of the Compiler Design Lab, which focused on developing a C language parser that utilizes the C lexer to parse input C files. In the previous phase, our efforts were concentrated on creating a lexical analyzer (flex script) to produce a stream of tokens from the source code and populate the symbol table. However, this lexical analyzer was primarily designed to handle errors related to invalid tokens.

In the C programming language, errors can also occur in the structure of the code, such as syntax errors and unbalanced parentheses. Therefore, it is essential to develop a parser capable of handling these types of errors. Following the lexical phase, the compiler progresses into the syntax analysis phase, and the parser plays a crucial role in this phase.

During syntax analysis, the compiler's primary task is to verify whether the tokens generated by the lexical analyzer adhere to the syntactic rules of the C language. This verification process is carried out by the parser, which takes a sequence of tokens from the lexical analyzer and ensures that this sequence conforms to the grammar rules of the source language. Additionally, the parser identifies and reports any syntax errors and generates a parse tree, which can be subsequently used to produce intermediate code.

Introduction:

Syntactic Analysis and Parser

The process of analyzing the syntax of a programming language involves two main stages. First, the lexical analyzer converts the source code into a stream of tokens, which are then stored in the symbol table. Second, the parser checks if the sequence of tokens can be derived from the grammar of the language. It identifies and reports various types of errors such as “structural errors”, “missing identifiers”, “wrong keywords”, and “unbalanced parentheses”.

The parser, also known as the syntax analyzer, is categorized into two types: top-down and bottom-up parsers.

The top-down parser expands non-terminals to generate the parse tree, starting from the initial symbol and ending with the terminals.

In contrast, the bottom-up parser compresses non-terminals, building the parse tree from non-terminals to the initial symbol, utilizing the reverse of the rightmost derivation.

YACC Script

Yacc, written in a C-based language, follows C's syntactic conventions and provides a powerful tool for organizing the structure of a program's input. The user defines rules describing the input structure, associated code for rule recognition, and a low-level routine for input handling. Yacc then generates a controlling function, the parser, which utilizes the user-defined lexical analyzer to extract tokens from the input stream. These tokens adhere to grammar rules, triggering user-defined actions when recognized. Yacc ensures early detection of input errors during a left-to-right scan, reducing the likelihood of working with faulty data. Error handling allows the program to resume processing after skipping over erroneous data. Yacc supports a broad class of input specifications, including LALR(1) grammars with disambiguating rules.

Below is a representation of the structure of a Yacc script:

```
declarations

%%

rules

%%

programs
```

The declaration section within a Yacc script is not mandatory. If it is absent, the second %% mark, which typically indicates the start of the program's main section, can also be omitted.

Spaces, tabs, and newlines are generally disregarded except when they are part of names or multi-character reserved symbols. Comments, enclosed within /* ... */, can be included anywhere a name is valid within the script, similar to the conventions followed in languages like C and PL/I.

The rules section constitutes one of the fundamental components of a Yacc script. It consists of one or more grammar rules. Each grammar rule follows a specific format, characterized by a set structure:

```
 A : BODY ;
```

'A' above refers to a name that represents a nonterminal in the grammar. 'BODY' denotes a sequence that can comprise zero or more names and literals. The colon ':' and the semicolon ';' are specific punctuation marks in Yacc that delineate the structure of grammar rules. Each rule can be associated with an action, which executes once all the symbols in the rule have been parsed. Actions essentially consist of C-program statements enclosed within curly braces '{}'.

C Program:

This section explains the C program that is supplied as input to the yacc script so that it can be parsed. Below is an explanation of the workflow:

1. To compile Yacc script.

```
yacc parser.y -d
```

2. To compile the flex script.

```
flex scanner.l
```

3. A lex.yy.c file is produced following the compilation of the lex file and compiling the yacc script generates the y.tab.c file.

4. The two files, lex.yy.c, y.tab.c are compiled together with the options -w.

```
gcc lex.yy.c y.tab.c -w
```

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

```
./a.out <filename.c>
```

In addition, standard input can be used by the script in place of file input.

Design of Programs:

Code:

scanner.l :

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

    #define ANSI_COLOR_RED      "\x1b[31m"
    #define ANSI_COLOR_GREEN    "\x1b[32m"
    #define ANSI_COLOR_YELLOW   "\x1b[33m"
```

```c
#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[1009];

struct constanttable
{
    char name[100];
    char type[100];
    int length;
}CT[1009];

int hash(char *str)
{
    int value = 0;
    for(int i = 0 ; i < strlen(str) ; i++)
    {
        value = 10*value + (str[i] - 'A');
        value = value % 1007;
        while(value < 0)
            value = value + 1007;
    }
    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)%1007)
        {
            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)%1007)
        {
            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)%1007)
        {
            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 < 1007 ; i++)
    {
        if(strcmp(ST[i].name,str1)==0)
        {
```

```c
                strcpy(ST[i].type,str2);
            }
        }
    }

    void insertSTvalue(char *str1, char *str2)
    {
        for(int i = 0 ; i < 1007 ; i++)
        {
            if(strcmp(ST[i].name,str1)==0)
            {
                strcpy(ST[i].value,str2);
            }
        }
    }

    void insertSTline(char *str1, int line)
    {
        for(int i = 0 ; i < 1007 ; 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)%1007)
            {
                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 < 1007 ; 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 < 1007 ; 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;}
"int*"           { strcpy(curtype,yytext); insertST(yytext,
"Keyword");return INTs;}
"float*"         { strcpy(curtype,yytext); insertST(yytext,
"Keyword");return FLOATs;}
"char*"          { strcpy(curtype,yytext); insertST(yytext,
"Keyword");return CHARs;}
"double*"        { strcpy(curtype,yytext); insertST(yytext,
"Keyword");return DOUBLEs;}
"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;}
"continue"       { insertST(yytext, "Keyword"); return CONTINUE;}
"switch"         { insertST(yytext, "Keyword"); return SWITCH;}
"case"           { insertST(yytext, "Keyword"); return CASE;}
"default"        { insertST(yytext, "Keyword"); return DEFAULT;}
"printf"         { insertST(yytext, "Keyword"); return PRINTF;}
"scanf"          { insertST(yytext, "Keyword"); return SCANF;}
"enum"           { insertST(yytext, "Keyword"); return ENUM;}
"union"          { insertST(yytext, "Keyword"); return UNION;}
"main"           { insertST(yytext, "Keyword"); return identifier;}



"++"             { 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-z|A-Z]([a-z|A-Z]|[0-9])*/\[  {strcpy(curid,yytext); insertST(yytext,
"Array Identifier");  return identifier;}
[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]
{strcpy(curval,yytext); insertCT(yytext, "Number Constant"); return
integer_constant;}
([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^]
{strcpy(curval,yytext); insertCT(yytext, "Floating Constant"); return
float_constant;}
[A-Za-z_][A-Za-z_0-9]*
{strcpy(curid,yytext);insertST(yytext,"Identifier");  return identifier;}

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

%%
```

parser.y :

```
%{
    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 UNION ENUM
%token RETURN MAIN
%token VOID
%token SWITCH CASE DEFAULT
%token WHILE FOR DO
%token PRINTF SCANF
%token BREAK CONTINUE
%token ENDIF
%token INTs FLOATs CHARs DOUBLEs
/* %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
            | union_definition
            | enum_declaration;



variable_declaration
            : type_specifier variable_declaration_list ';'
            | structure_declaration
            | union_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
            : '[' initilization_params
            | ;

initilization_params
            : integer_constant ']' initilization
            | ']' string_initilization;

initilization
            : string_initilization
            | array_initialization
            | ;

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

star_specifier
            : INTs | CHARs | FLOATs | DOUBLEs ;

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  '}' ';';

union_definition
            : UNION identifier { ins(); } '{' V1  '}' ';';

enum_declaration
            : ENUM identifier '{' enum_list '}' ';'
            ;

enum_list
            : enumerator
            | enum_list ',' enumerator ;

enumerator
            : identifier
            | identifier assignment_operator integer_constant
            ;

V1 : variable_declaration V1 | ;

structure_declaration
            : STRUCT identifier variable_declaration_list;

union_declaration
            : UNION 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_statment | compound_statement
            | conditional_statements | iterative_statements
            | return_statement | break_statement
            | continue_statement | switch_statement
            | printf_scanf_statements
            | variable_declaration ;

compound_statement
            : '{' statment_list '}' ;

statment_list
            : statement statment_list
            | ;

printf_scanf_statements
            : printf_statement ';'
            | scanf_statement ';'
            ;
```

```
printf_statement
            : PRINTF '(' printf_parameters ')' ;

scanf_statement
            :  SCANF '(' scanf_parameters ')' ;

printf_parameters
    : string_constant ',' expression
    | string_constant;

scanf_parameters
    : string_constant ',' '&' identifier
    | string_constant;

expression_statment
            : expression ';'
            | ';' ;

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

conditional_statements_breakup
            : ELSE statement
            | ;

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

return_statement
            : RETURN return_statement_breakup;

return_statement_breakup
            : ';'
            | expression ';' ;

break_statement
            : BREAK ';' ;

continue_statement
            : CONTINUE ';' ;

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
            | ;

switch_statement
            : SWITCH '(' simple_expression ')' '{' switch_body '}' ;

switch_body
            : case_list default_case
            | case_list
            | default_case ;

case_list
            : case_clause
            | case_list case_clause ;

case_clause
            : CASE
            | constant ':' statement
            | constant ':' statement break_statement;

default_case
            : DEFAULT  ':' statement
            | DEFAULT  ':' statement break_statement ;


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 *);
```

```c
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;
}
```

Explanation :

This report outlines the progress made during Phase Two of the Compiler Design Lab, which focused on developing a C language parser that utilizes the C lexer to parse input C files. In the previous phase, our efforts were concentrated on creating a lexical analyzer (flex script) to produce a stream of tokens from the source code and populate the symbol table. However, this lexical analyzer was primarily designed to handle errors related to invalid tokens.

In the C programming language, errors can also occur in the structure of the code, such as syntax errors and unbalanced parentheses. Therefore, it is essential to develop a parser capable of handling these types of errors. Following the lexical phase, the compiler progresses into the syntax analysis phase, and the parser plays a crucial role in this phase.

During syntax analysis, the compiler's primary task is to verify whether the tokens generated by the lexical analyzer adhere to the syntactic rules of the C language. This verification process is carried out by the parser, which takes a sequence of tokens from the lexical analyzer and ensures that this sequence conforms to the grammar rules of the source language. Additionally, the parser identifies and reports any syntax errors and generates a parse tree, which can be subsequently used to produce intermediate code.

Declaration Section:

The declaration section encompasses essential elements such as header files, function declarations, and flags necessary for the code. It serves as the preparatory part of the code, ensuring that all required components are in place. Additionally, within the declaration section, tokens returned by the lexer are listed in order of precedence. Operators are also declared, along with their associativity and precedence, which is critical for ensuring that the grammar provided to the parser is unambiguous. This is particularly important because LALR(1) parsers cannot work effectively with ambiguous grammars.

Rules Section:

The rules section is where the production rules for the entire C language are documented. These rules are formulated in a manner that eliminates left recursion and ensures that the grammar is deterministic. To make the grammar deterministic, any instances of non-deterministic grammar are transformed through left factoring. This transformation is carried out to make the grammar suitable for an LL(1) parser. It's worth noting that LL(1) grammars are a subset of LALR(1) grammars, and this adjustment ensures that the grammar is well-structured and can be effectively parsed.

The primary role of the grammar productions in the rules section is to perform syntax analysis on the source code. When the parser successfully matches a complete statement with the correct syntax, it indicates that the source code adheres to the defined grammar rules and is ready for further processing.

<u>C program section:</u>

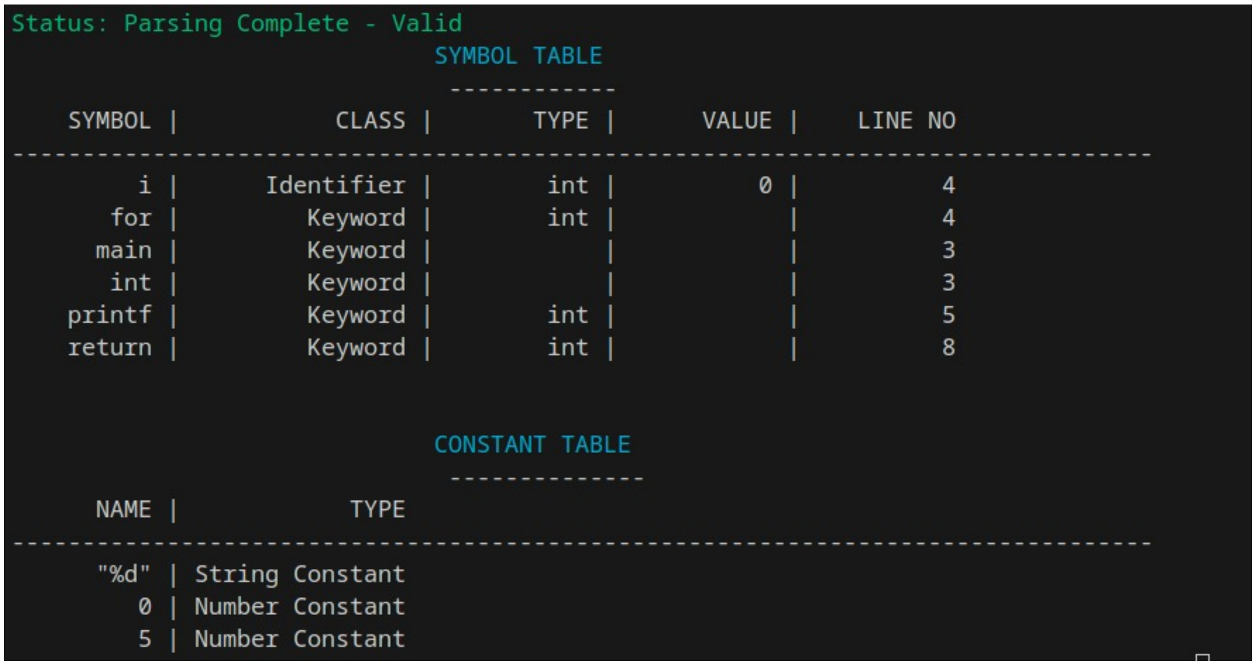
The C-Program Section serves as a pivotal component in the compiler's architecture, facilitating seamless communication between the parser and the lexer. It establishes connections with functions and variables declared in the lexer, manages external files generated by the lexer, and orchestrates the compilation process via the main function, which takes the input source code file and coordinates the execution of the compiler. A key outcome of this section is the generation and printing of the final symbol table, encapsulating comprehensive information about identifiers, variables, and functions in the source code, thus marking the successful completion of the compilation process.

<u>Test Cases</u> :

Test Case 1 :

```
#include<stdio.h>

int main() {
    for (int i = 0; i < 5; i++) {
        printf("%d", i);
    }

    return 0;
}
```

Output 1 :

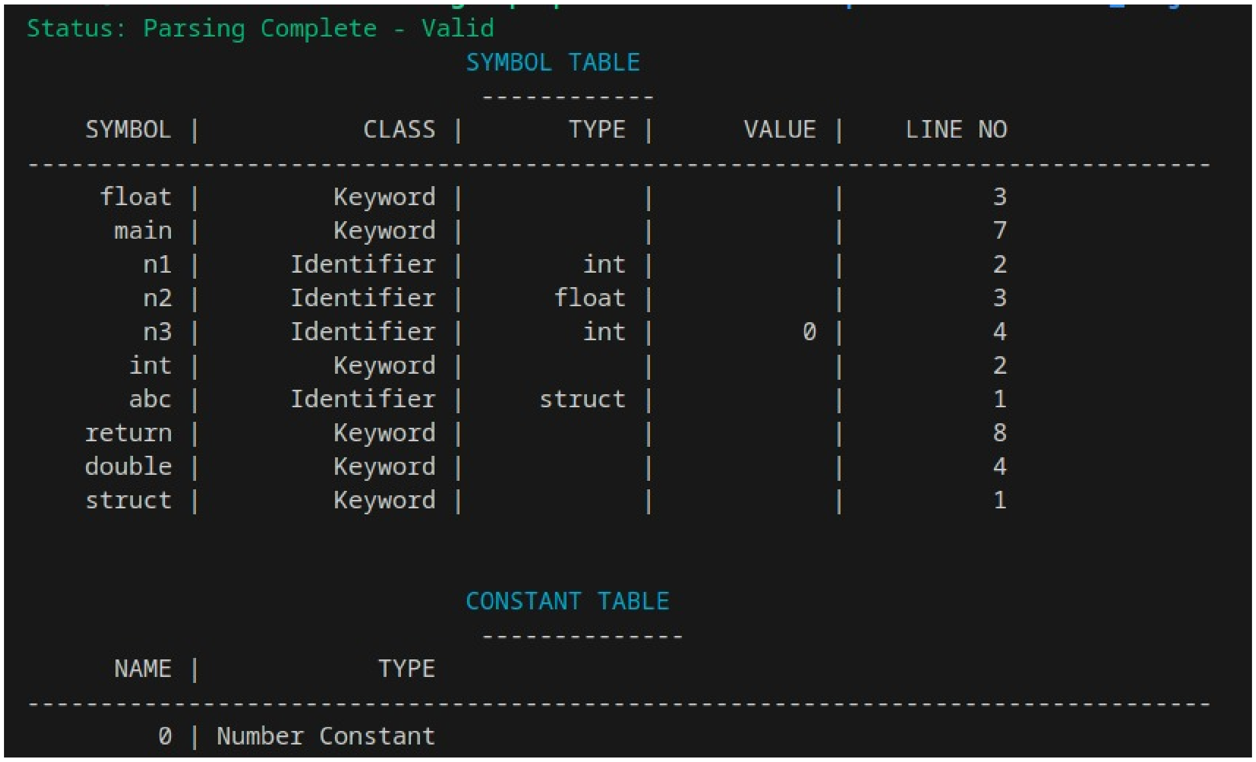


```
Status: Parsing Complete - Valid
                              SYMBOL TABLE
                               ------------
    SYMBOL |           CLASS |        TYPE |       VALUE |     LINE NO
---------------------------------------------------------------------------------
         i |      Identifier |         int |           0 |           4
       for |         Keyword |         int |             |           4
      main |         Keyword |             |             |           3
       int |         Keyword |             |             |           3
    printf |         Keyword |         int |             |           5
    return |         Keyword |         int |             |           8


                              CONSTANT TABLE
                               --------------
      NAME |            TYPE
---------------------------------------------------------------------------------
      "%d" | String Constant
         0 | Number Constant
         5 | Number Constant
```

Test Case 2 :

```c
struct abc {
    int n1;
    float n2;
    double n3;
};

int main() {
    return 0;
}
```

Output 2 :

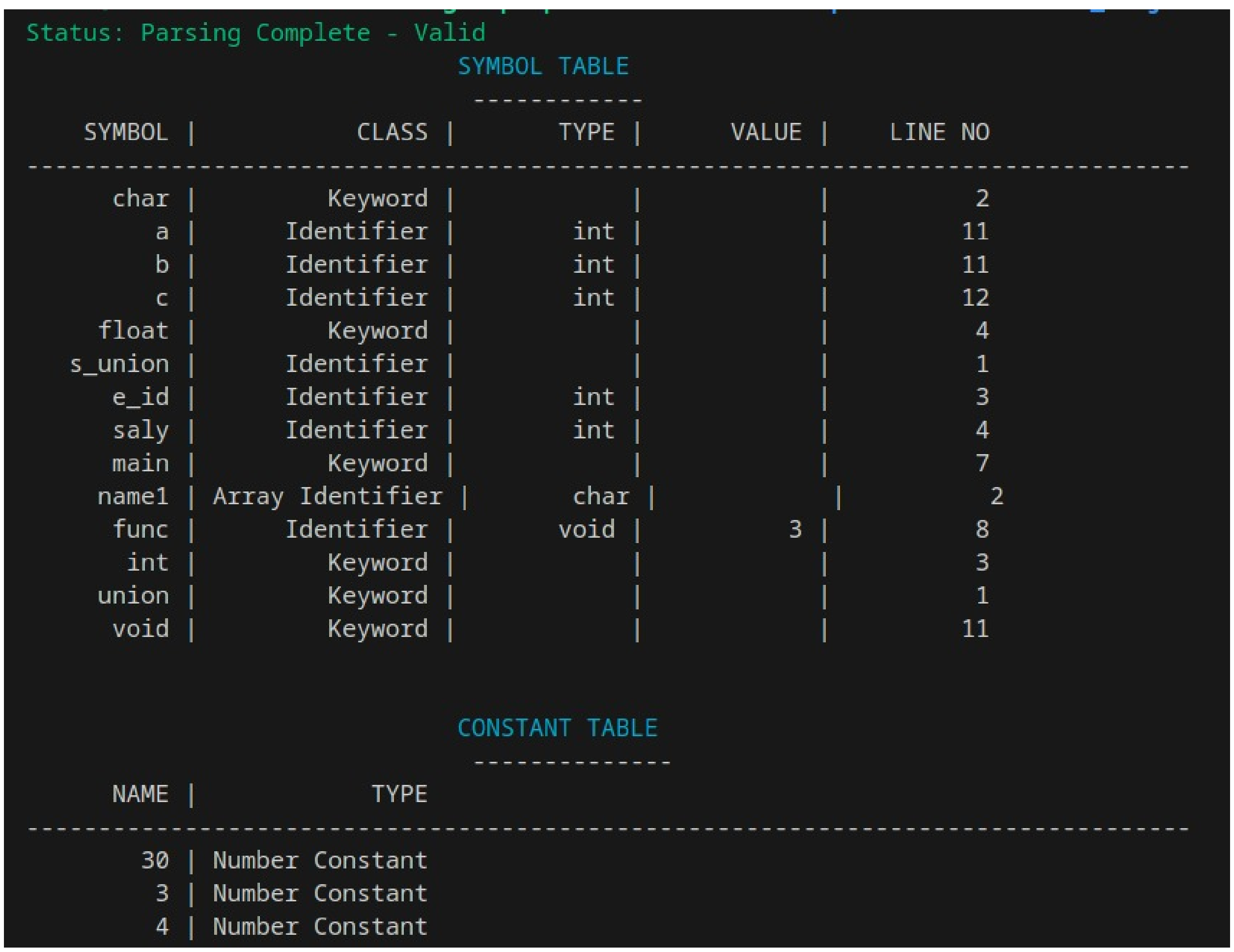


```
Status: Parsing Complete - Valid
                                   SYMBOL TABLE
                                   ------------
    SYMBOL |           CLASS |         TYPE |        VALUE |     LINE NO
-----------------------------------------------------------------------------------
     float |         Keyword |              |              |           3
      main |         Keyword |              |              |           7
        n1 |      Identifier |          int |              |           2
        n2 |      Identifier |        float |              |           3
        n3 |      Identifier |          int |            0 |           4
       int |         Keyword |              |              |           2
       abc |      Identifier |       struct |              |           1
    return |         Keyword |              |              |           8
    double |         Keyword |              |              |           4
    struct |         Keyword |              |              |           1


                                   CONSTANT TABLE
                                   --------------
      NAME |            TYPE
-----------------------------------------------------------------------------------
         0 | Number Constant
```

Test Case 3:

```c
union s_union {
    char name1[30];
    int e_id;
    float saly;
};

int main() {
    func(4, 3);
}

void func(int a, int b) {
    int c = a + b;
}
```

Output 3:

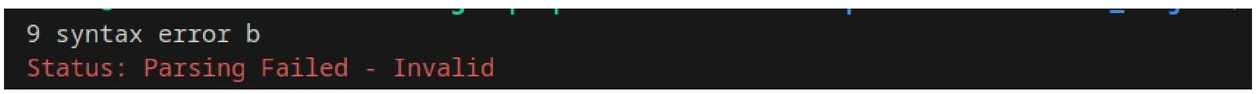


```
Status: Parsing Complete - Valid
                              SYMBOL TABLE
                               ------------
    SYMBOL |             CLASS |        TYPE |       VALUE |     LINE NO
--------------------------------------------------------------------------------
      char |           Keyword |             |             |           2
         a |        Identifier |         int |             |          11
         b |        Identifier |         int |             |          11
         c |        Identifier |         int |             |          12
     float |           Keyword |             |             |           4
   s_union |        Identifier |             |             |           1
      e_id |        Identifier |         int |             |           3
      saly |        Identifier |         int |             |           4
      main |           Keyword |             |             |           7
     name1 | Array Identifier |        char |             |           2
      func |        Identifier |        void |           3 |           8
       int |           Keyword |             |             |           3
     union |           Keyword |             |             |           1
      void |           Keyword |             |             |          11


                              CONSTANT TABLE
                               --------------
      NAME |              TYPE
--------------------------------------------------------------------------------
        30 | Number Constant
         3 | Number Constant
         4 | Number Constant
```

Test Case 4:

```c
#include<stdio.h>

int main() {
    int a = func(4, 3);

    return 0;
}

int func(int a, b) {
    return a + b;
}
```

Output 4:

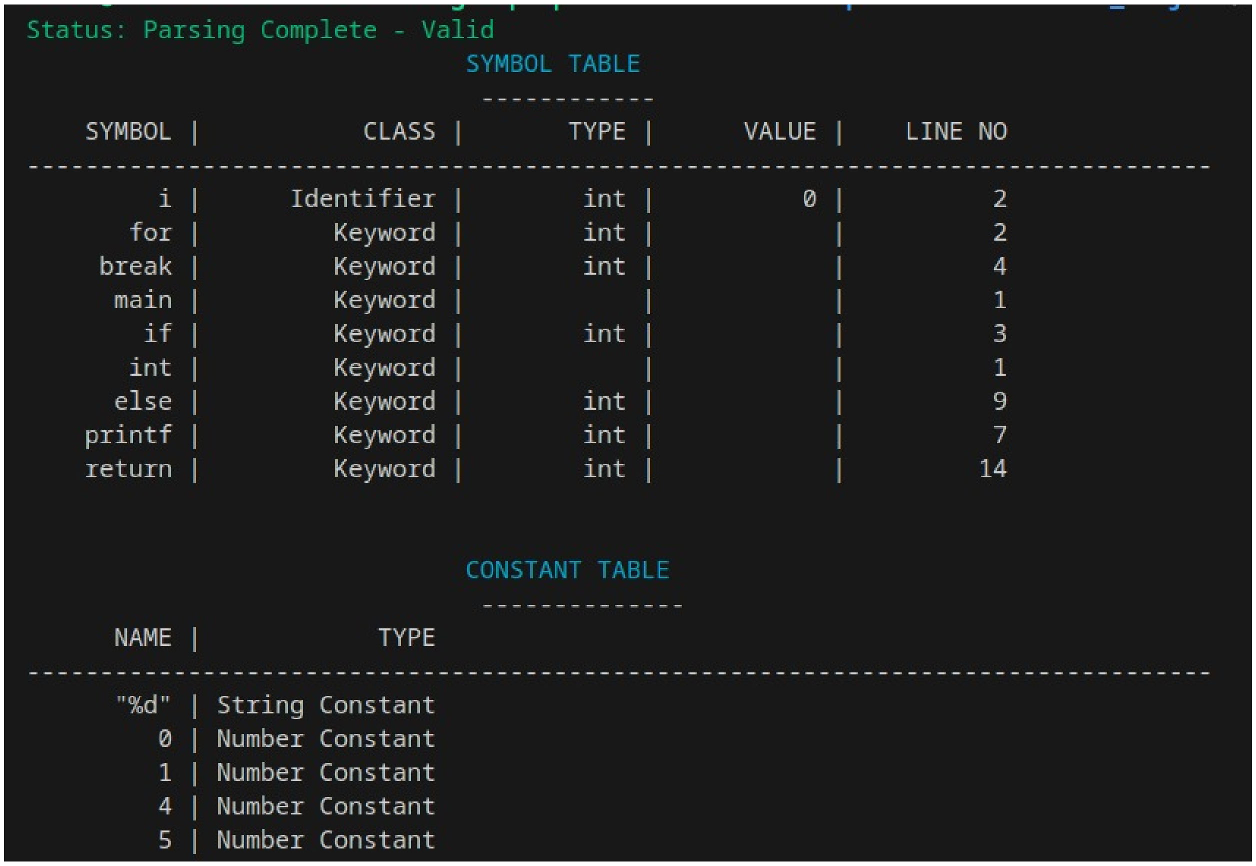


```
9 syntax error b
Status: Parsing Failed - Invalid
```

Test Case 5:

```
int main() {
    for (int i = 0; ; i++) {
        if (i > 5) {
            break;
        }
        else if (i < 4) {
            printf("%d", i);
        }
        else {
            printf("%d", i-1);
        }
    }

    return 0;
}
```

Output 5:

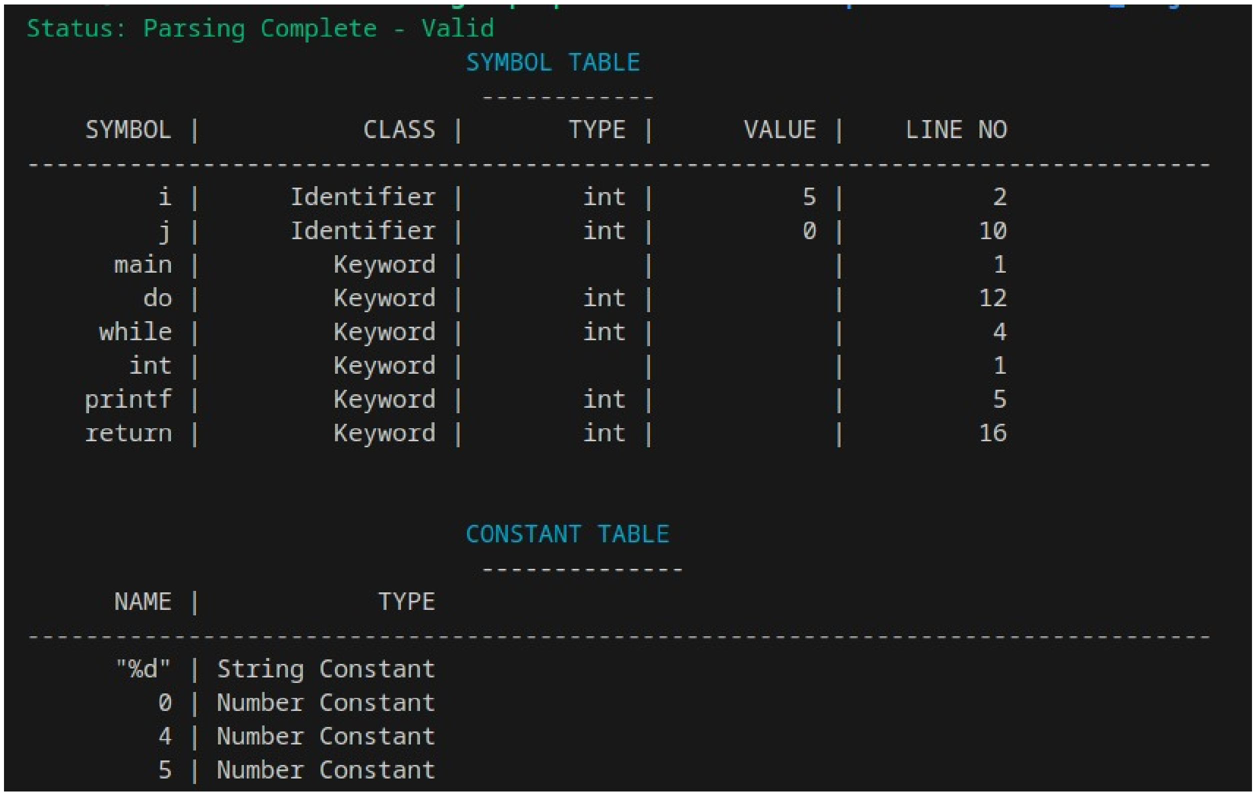


```
Status: Parsing Complete - Valid
                              SYMBOL TABLE
                               ------------
     SYMBOL |          CLASS |       TYPE |       VALUE |     LINE NO
--------------------------------------------------------------------------------
          i |     Identifier |        int |           0 |           2
        for |        Keyword |        int |             |           2
      break |        Keyword |        int |             |           4
       main |        Keyword |            |             |           1
         if |        Keyword |        int |             |           3
        int |        Keyword |            |             |           1
       else |        Keyword |        int |             |           9
     printf |        Keyword |        int |             |           7
     return |        Keyword |        int |             |          14


                              CONSTANT TABLE
                               --------------
       NAME |           TYPE
--------------------------------------------------------------------------------
       "%d" | String Constant
          0 | Number Constant
          1 | Number Constant
          4 | Number Constant
          5 | Number Constant
```

Test Case 6:

```c
int main() {
    int i = 0;

    while (i < 5) {
        printf("%d", i);

        i++;
    }

    int j = 0;

    do {
        printf("%d", i+j);
    } while (j < 4);

    return 0;
}
```

Output 6:

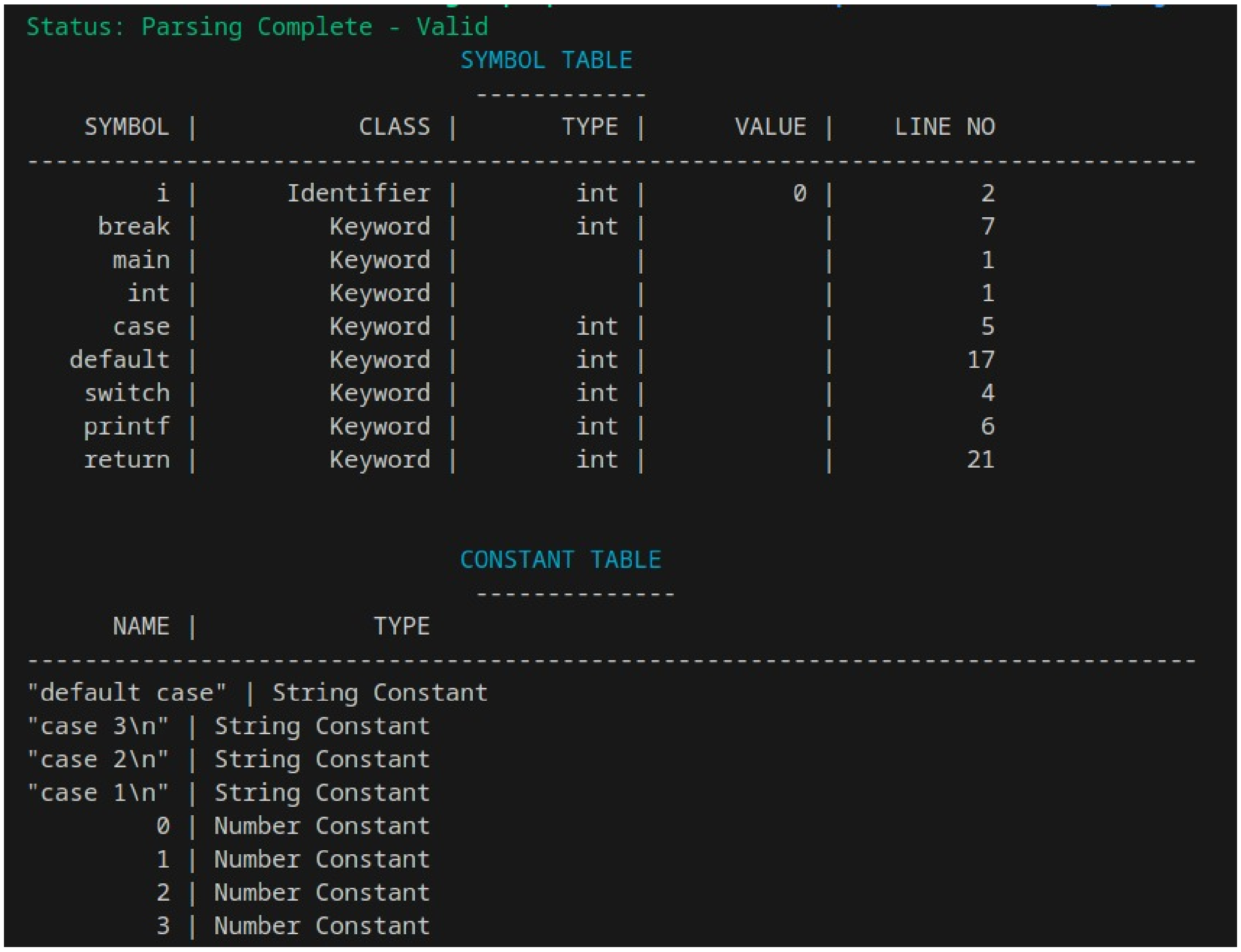


```
Status: Parsing Complete - Valid
                              SYMBOL TABLE
                               ------------
    SYMBOL |          CLASS |       TYPE |      VALUE |    LINE NO
---------------------------------------------------------------------------------
         i |     Identifier |        int |          5 |          2
         j |     Identifier |        int |          0 |         10
      main |        Keyword |            |            |          1
        do |        Keyword |        int |            |         12
     while |        Keyword |        int |            |          4
       int |        Keyword |            |            |          1
    printf |        Keyword |        int |            |          5
    return |        Keyword |        int |            |         16


                              CONSTANT TABLE
                               --------------
      NAME |           TYPE
---------------------------------------------------------------------------------
      "%d" | String Constant
         0 | Number Constant
         4 | Number Constant
         5 | Number Constant
```

Test Case 7:

```c
int main() {
    int i = 0;

    switch(i+2) {
        case 1:
            printf("case 1\n");
            break;

        case 2:
            printf("case 2\n");
            break;

        case 3:
            printf("case 3\n");
            break;

        default:
            printf("default case");
    }

    return 0;
}
```

Output 7:

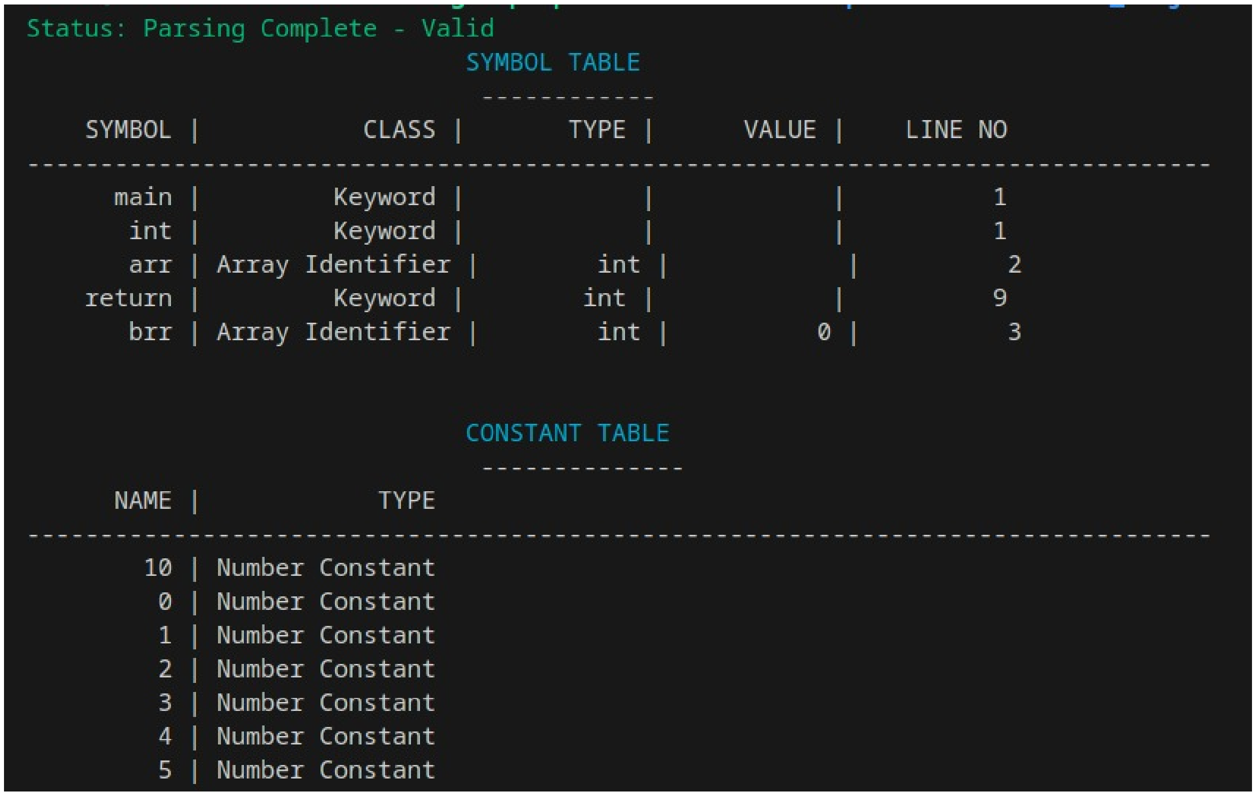


```
Status: Parsing Complete - Valid
                              SYMBOL TABLE
                               ------------
    SYMBOL |             CLASS |        TYPE |        VALUE |     LINE NO
---------------------------------------------------------------------------------
         i |        Identifier |         int |            0 |           2
     break |           Keyword |         int |              |           7
      main |           Keyword |             |              |           1
       int |           Keyword |             |              |           1
      case |           Keyword |         int |              |           5
   default |           Keyword |         int |              |          17
    switch |           Keyword |         int |              |           4
    printf |           Keyword |         int |              |           6
    return |           Keyword |         int |              |          21


                              CONSTANT TABLE
                               --------------
      NAME |             TYPE
---------------------------------------------------------------------------------
"default case" | String Constant
"case 3\n" | String Constant
"case 2\n" | String Constant
"case 1\n" | String Constant
         0 | Number Constant
         1 | Number Constant
         2 | Number Constant
         3 | Number Constant
```

Test Case 8:

```
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int brr[10];

    //single line comment

    /* multiline
        comment */

    return 0;
}
```

Output 8:

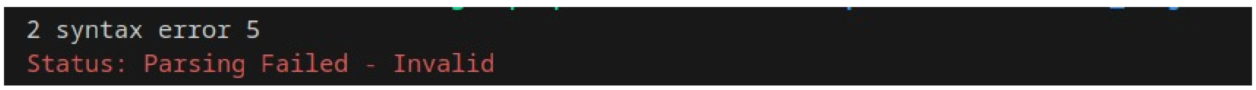


```
 Status: Parsing Complete - Valid
                                  SYMBOL TABLE
                                   ------------
     SYMBOL |           CLASS |       TYPE |      VALUE |     LINE NO
 --------------------------------------------------------------------------------
       main |         Keyword |            |            |          1
        int |         Keyword |            |            |          1
        arr | Array Identifier |        int |            |          2
     return |         Keyword |        int |            |          9
        brr | Array Identifier |        int |          0 |          3


                                  CONSTANT TABLE
                                   --------------
       NAME |            TYPE
 --------------------------------------------------------------------------------
         10 | Number Constant
          0 | Number Constant
          1 | Number Constant
          2 | Number Constant
          3 | Number Constant
          4 | Number Constant
          5 | Number Constant
```

Test Case 9:

```c
int main() {
    printf(5);
}
```

Output 9:

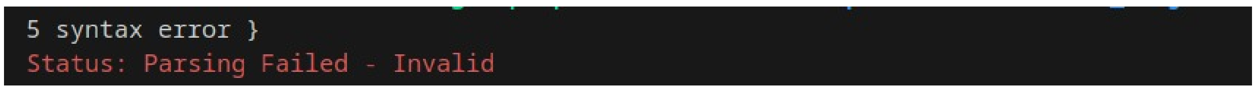


```
2 syntax error 5
Status: Parsing Failed - Invalid
```

Test Case 10:

```c
int main() {

    //missing semi-colon
    int a
}
```

Output 10:

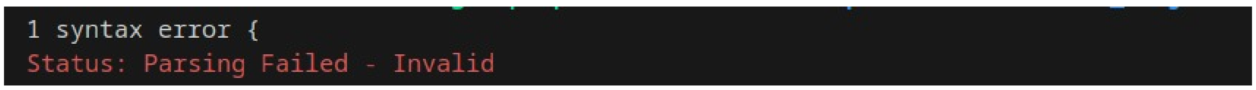


```
5 syntax error }
Status: Parsing Failed - Invalid
```

Test Case 11:

```
int main {
    int a;
}
```

Output 11:

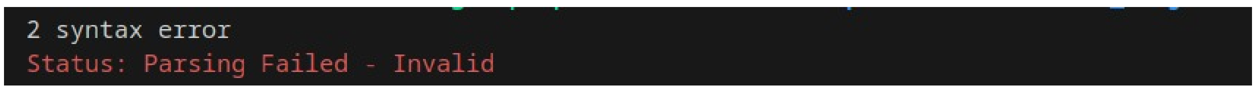


```
1 syntax error {
Status: Parsing Failed - Invalid
```

Test case 12:

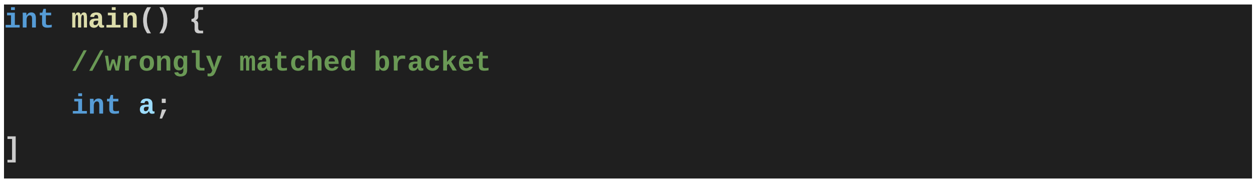
```
int main() {
    int a;
```

Output 12:



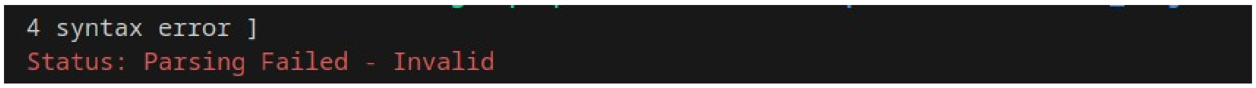
```
2 syntax error
Status: Parsing Failed - Invalid
```

Test Case 13:



```
int main() {
    //wrongly matched bracket
    int a;
]
```

Output 13:



```
4 syntax error ]
Status: Parsing Failed - Invalid
```

<u>Assumption:</u>

We have assumed the existence of at least one function in the code. A code without function will give a syntax error, even if the code is syntactically correct. So, it is advisable to make one function to run the code for syntax analysis. This is due to the way we have written the production rules in our parser.

It is also assumed that the struct, union and enum functions are declared outside a function. This is due to the way the production rules have been written by us.

<u>Implementation:</u>

In the previous phase, the lexer code was designed to handle most features of the C language using regular expressions. However, some specific cases required custom regular expressions to be addressed. These cases included handling identifiers, supporting multiline comments, managing literals, ensuring error handling for incomplete strings, and managing nested comments.

To ensure exhaustive token recognition, the parser code integrated the lexer code designed for the C language specifications. The parser implemented the C grammar by utilizing various production rules. It consumed tokens from the lexer's output, one at a time, and applied the corresponding production rules to update the symbol table with the relevant type, value, and line of declaration information. If parsing encountered an issue, the parser flagged the specific line number along with the corresponding error message.

To maintain the symbol table effectively, the following functions were implemented:

1. **LookupST()**: This function checked whether a token already existed in the symbol table. If it did, the function returned 1; otherwise, it returned 0. (Called by the Scanner)
2. **InsertST()**: This function added the token to the symbol table, along with the token class, if it was not already present. (Called by the Scanner)
3. **InsertSTtype()**: This function appended the data type of the identifier to the symbol table. (Called by the Parser)

4. **InsertSTvalue()**: This function appended the value of the identifier to the symbol table. (Called by the Parser)
5. **InsertSTline()**: This function appended the line of the declaration of the identifier to the symbol table. (Called by the Parser)
6. **LookupCT()**: This function checked whether a token already existed in the constant table. If it did, the function returned 1; otherwise, it returned 0. (Called by the Scanner)
7. **InsertCT()**: This function added the token to the constant table, along with the token class, if it was not already present. (Called by the 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.

<u>Result:</u>

The Yacc script effectively parsed all the tokens generated by the Flex script for the C programming language. For all the identifiers and constants within the program, the script successfully returned the type, value, and line of declaration as the output. In the case of encountering errors, the script also returned the line number along with an appropriate syntax error message. In summary, the provided Yacc script demonstrated the capability to parse tokens and generate relevant error messages when handling C programs.

Future Scope:

The provided YACC script is designed to analyze certain rules of the C language. While it covers several syntaxes, there are still many aspects of the C language that it does not support. The plan is to enhance the script to encompass a more comprehensive set of syntaxes and rules in the future. The ultimate goal is to create a complete compiler that can handle all aspects of the C language by providing exhaustive rule support for every C language syntax.