CS304- COMPILER DESIGN LAB

A REPORT ON THE PROJECT ENTITLED

SYNTAX ANALYZER FOR THE C LANGUAGE

Group Members:

B Anagha	Gouri M R	P Devi Deepika
221CS114	221CS125	221CS138

V SEMESTER B-TECH CSE- S1

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL

2024 – 2025

Abstract

A compiler is a special program that processes statements written in a particular programming language (high-level language) and turns them into machine language (low-level language) that a computer's processors use. Apart from this, the compiler is also responsible for detecting and reporting any errors in the source program during the translation process.
The file used for writing code in a specific language contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.
This report specifies the details related to the second stage of the compiler, the parsing stage. We have developed a parser for the C programming language using the lex and yacc tools. The parser makes use of the tokens outputted by the lexer developed in the previous stage to parse the C input file. The lexical analyzer can detect only lexical errors like unmatched comments etc. but cannot detect syntactical errors like missing semi-colon etc. These syntactical errors are identified by the parser i.e. the syntax analysis phase is done by the parser.

Introduction

Syntax Analysis

Syntax analysis is the second phase of the compiler design process which follows the lexical analysis phase. The parser takes as input the stream of tokens we get from the lexical analysis stage. It analyses the syntactical structure of the given input i.e. it verifies that a string of token names can be generated by the grammar of the source language. It checks if the given input is in the correct syntax of the programming language in which the input which has been written with the help of a Parse Tree or Syntax Tree.

The Parse Tree is developed with the help of pre-defined grammar of the language. The syntax analyzer also checks whether a given program fulfills the rules implied by a context-free grammar. If it satisfies, the parser then creates the parse tree of that source program. In the case of an invalid grammar, we expect the parser to report the appropriate syntax errors in a neat and intelligible manner. The errors identified and reported by the parser include:

• Unbalanced Parenthesis

• Missing Semi-colons

• Errors in Structure

• Missing Operators

• Misspelt Keywords

Parsing Techniques

Parsing techniques, in general, can be divided into two different groups.

• Top Down Parsing

• Bottom Up Parsing

Top Down Parsing can further be divided into **predictive parsing** and **recursive descent parsing. Predictive parse** can predict which production should be used to replace the specific input string. The predictive parser uses look-ahead point, which points towards next input symbols. Backtracking is not an issue with this parsing technique. It is known as LL(1) Parser. The **recursive descent parsing** technique

recursively parses the input to make a parse tree. It consists of several small functions, one for each nonterminal in the grammar.

In the **Bottom Up Parsing** technique, the construction of the parse tree starts witH the leaf nodes, and then it processes towards its root. It is also called as shift-reduce parsing. This type of parsing is created with the help of using some software tools.

Yacc Script

Yacc stands for Yet Another Compiler-Compiler. Yacc is essentially a parser generator. Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. A function is then generated by Yacc to control the input process. This function is called the parser which calls the lexical analyzer to get a stream of tokens from the input. Based on the input structure rules, called grammar rules, the tokens are organized. When one of these rules has been recognized, then user code supplied for this rule, an action, is invoked. Actions have the ability to return values and make use of the values of other actions.

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 divided into three sections, separated by lines that contain only two percent signs, as follows:

DECLARATIONS

%%

RULES

%%

AUXILIARY FUNCTIONS

The **Declarations Section** defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied directly into the generated source file. We also define all parameters related to the parser here, specifications like using leftmost derivations or rightmost derivations, precedence, left and right associativity are declared here, data types and tokens which will be used by the lexical analyzer are also declared at this stage.

The **Rules Section** contains the entire grammar which is used for deciding if the input text is legally correct according to the specifications of the language. Yacc uses these rules for reducing the token stream received from the lexical analysis stage. All rules are linked to each other from the start state. Yacc generates C code for the rules specified in the Rules section and places this code into a single function called yyparse(). The Auxiliary Functions Section contains C statements and functions that are copied directly to the generated source file. These statements usually contain code called by the different rules. This section essentially allows the programmer to add to the generated source code.

C Program

The parser takes C source files as input for parsing. The input file is specified in the auxiliary functions section of the yacc script. The workflow for testing the parser is as follows:

1. Compile the yacc script using the yacc tool

$ yacc -d parser.y

2. Compile the flex script using the flex tool

$ lex lexer.l

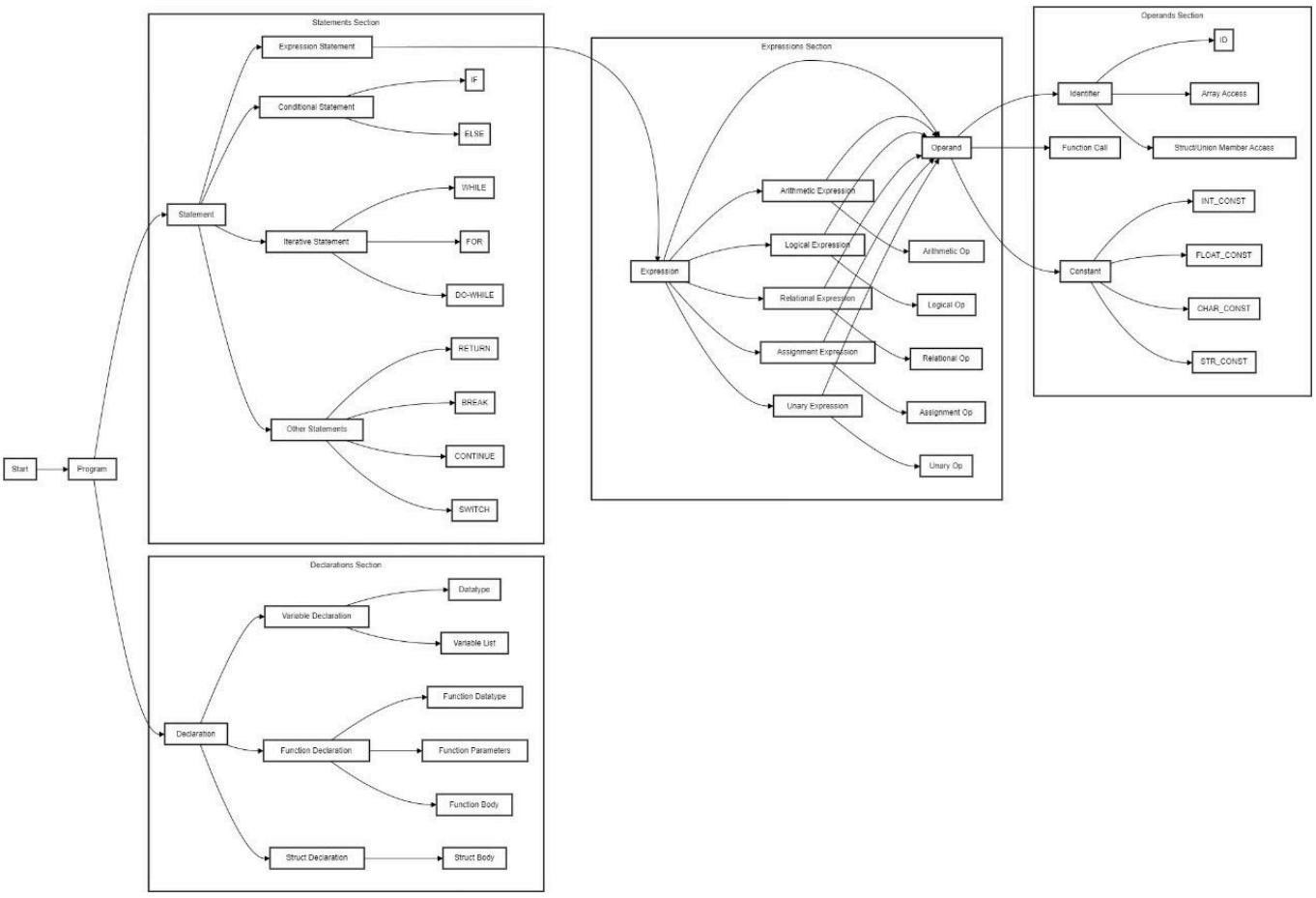
3. The first two steps generate lex.yy.c, y.tab.c, and y.tab.h. The header file is

included in lexer.l file. Then, lex.yy.c and y.tab.c are compiled together.

$ gcc lex.yy.c y.tab.c

4. Run the generated executable file

$ ./a.out



FLOWCHART OF SYNTAX ANALYSIS

GRAMMAR BEING PARSED:

S -> content

content -> declaration content
| ε

declaration -> variable_declaration
| function_declaration
| struct_dec

struct_dec -> STRUCT_UNION ID '{' struct_body '}' ';'

struct_body -> variable_declaration struct_body
| ε

```
variable_declaration -> datatype var ';'
                    | struct_initialise

struct_initialise -> STRUCT_UNION ID var
                   | STRUCT_UNION ID star var

var -> id_name
   | id_name ',' var

id_name -> ID extended_id

extended_id -> arr_id
            | '=' exp
            | ε

arr_id -> '[' INT_CONST ']' initialise
       | '[' INT_CONST ']' arr_id

initialise -> string_initialise
            | arr_initialise
            | ε

string_initialise -> '=' STR_CONST

arr_initialise -> '=' multi_dim

multi_dim -> '{' arr_elements '}'

arr_elements -> arr_values
              | '{' arr_values '}'
              | multi_dim
              | multi_dim ',' '{' arr_elements '}'
              | '{' arr_values '}' ',' arr_elements

arr_values -> constant multiple_arr_values

multiple_arr_values -> ',' arr_values
                     | ε

datatype -> INT
          | DATA_TYPE
          | SIZE_MODIFIER grammar
          | SIGN sign_grammar
```

```
        | SCOPE scope_grammar
        | datatype star

star -> '*' star
     | '*'

scope_grammar -> INT
              | DATA_TYPE
              | SIGN grammar
              | ε

sign_grammar -> INT
             | SIZE_MODIFIER grammar
             | ε

grammar -> INT
        | ε

function_declaration -> fun_datatype fun_param

fun_datatype -> datatype ID '('

fun_param -> param ')' statement

param -> datatype all_param_id
      | ε

all_param_id -> param_id multiple_param

multiple_param -> ',' param
               | ε

param_id -> ID ext_param

ext_param -> '[' ']'
          | ε

statement -> expression_statement
          | multiple_statements
          | conditional_statement
          | iterative_statement
          | return_statement
          | break_statement
          | continue_statement
```

```
        | switch_statement
        | variable_declaration

multiple_statements -> '{' block '}'

block -> statement block
      | ε

expression_statement -> exp ';'
                      | exp ',' expression_statement
                      | ';'

conditional_statement -> IF '(' simple_exp ')' statement
extended_conditional_statement

extended_conditional_statement -> ELSE statement
                                | ε

iterative_statement -> WHILE '(' simple_exp ')' statement
                     | FOR '(' for_initialise simple_exp ';' exp ')'
                     | DO statement WHILE '(' simple_exp ')' ';'

switch_statement -> SWITCH '(' simple_exp ')' '{' case_st DEFAULT ':' statement '}'

case_st -> CASE int_char_const ':' statement BREAK ';' case_st
        | ε

int_char_const -> INT_CONST
                | CHAR_CONST

for_initialise -> variable_declaration
                | exp ';'
                | ';'

return_statement -> RETURN return_suffix

return_suffix -> ';'
               | exp ';'

break_statement -> BREAK ';'

continue_statement -> CONTINUE ';'

exp -> identifier expression
```

```
   | simple_exp

expression -> '=' exp
          | OP_EQUAL exp
          | INC_DEC

simple_exp -> unary_relation_exp rel_exp_breakup

rel_exp_breakup -> LOG unary_relation_exp rel_exp_breakup
               | ε

unary_relation_exp -> NOT unary_relation_exp
                   | regular_exp

regular_exp -> arithmetic_exp regular_exp_breakup

regular_exp_breakup -> REL arithmetic_exp
                    | ε

arithmetic_exp -> arithmetic_exp operators factor
               | factor

operators -> '+'
          | '-'
          | '*'
          | '/'
          | '^'
          | '%'

factor -> fun
       | identifier

identifier -> ID
           | '&' ID
           | identifier ext_identifier

ext_identifier -> '[' exp ']'
               | '.' ID
               | "->" ID

fun -> '(' exp ')'
    | fun_call
    | constant
```

fun_call -> ID '(' arg ')'

arg -> arg_list
    | ε

arg_list -> exp ext_arg

ext_arg -> ',' exp ext_arg
        | ε

constant -> INT_CONST
         | STR_CONST
         | FLOAT_CONST
         | CHAR_CONST

CODE:

LEX Code

```
%option yylineno
 /* Declaration of variables and functions */
%{
#include <stdio.h>
#include <string.h>
#include "parser.tab.h"
int flag;

typedef struct node {
    char *key;
    char *value;
    struct node *next;
} token;

typedef struct symbol {
    char *name;                // Symbol name
    char *type;                // Data type
    char *class;               // Class (e.g., variable,function)
    int *boundaries_list;       // Array of boundaries
    int array_dimensions;       // Number of dimensions forarrays
    char **parameters_list;     // List of parameters for functions
    int procedure_definition;  // Flag: 1 if procedure, 0 otherwise
    int nesting_level;          // Scope level
    int line_declared;          // Line number where declared
    struct symbol *next;
} symbol;

token *constant_head = NULL;

int lineNo = 1;
int nestLevel = 0;
char currIdenName[100];
char currData[100];
```

```
extern symbol *symbol_table;
void add_symbol(char* name, char* type, char* class, int* boundaries, int arr_dim, char**
params, int proc_def, int nest_level, int line_no);

void addConstant(char* tok, char* type);

%}
 /* Regular definitions */
num [0-9]
alpha [a-zA-Z]
alphanum {alpha}|{num}
escape_sequences \\(0|a|b|f|n|r|t|v|"\\"|"\""|')
ws [ \t\r\f\v]+
%x ML
 /* Pattern Matching Rules */
%%
\n {lineNo++;}
"#include"[ ]*"<"{alpha}({alphanum})*".h>" { }
"#define"[ ]+(_|{alpha})({alphanum})*[ ]*(.)+ { }
"//".* { }
"/*"([^*]|\*+[^*/])*\*+"/" { }
"["|"]"|"("|")"|","|";"|":"|"."
{add_symbol(yytext,"NULL","Delimiter",NULL,0,NULL,0,nestLevel,lineNo); return *yytext;}
"{"    {nestLevel++; add_symbol("{","NULL","Delimiter",NULL,0,NULL,0,nestLevel,lineNo); return
*yytext;}
"}"    {nestLevel--; add_symbol("}","NULL","Delimiter",NULL,0,NULL,0,nestLevel,lineNo);return
*yytext;}
"->"|"+"|"-"|"*"|"/"|"="|"%"|"&"|"^"|","|":"|"?"|";" {
add_symbol(yytext,"NULL","Operator",NULL,0,NULL,0,nestLevel,lineNo); return *yytext; }
"++"|"--" { add_symbol(yytext,"NULL","Operator",NULL,0,NULL,0,nestLevel,lineNo); return
INC_DEC; }
"!" {  add_symbol(yytext,"NULL","Operator",NULL,0,NULL,0,nestLevel,lineNo); return NOT; }
```

```
"+="|"-="|"*="|"/="|"%=" {
add_symbol(yytext,"NULL","Operator",NULL,0,NULL,0,nestLevel,lineNo); return OP_EQUAL; }
"&&"|"||" {  add_symbol(yytext,"NULL","Operator",NULL,0,NULL,0,nestLevel,lineNo); return LOG; }
">="|"<="|">"|"<"|"=="|"!=" {
add_symbol(yytext,"NULL","Operator",NULL,0,NULL,0,nestLevel,lineNo); return REL; }
"int" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo);
strcpy(currData,"Int");return INT; }
"char"|"double"|"float"|"void" {
add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); strcpy(currData,yytext);
return DATA_TYPE; }
"long"|"short" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return
SIZE_MODIFIER; }
"signed"|"unsigned" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo);
return SIGN; }
"const"|"register"|"static"|"auto"|"extern" {
add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return SCOPE; }
"if" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return IF; }
"else" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return ELSE; }
"for" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return FOR; }
"while" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return WHILE; }
"do" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return DO; }
"struct"|"union" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo);return
STRUCT_UNION; }
"return" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return RETURN; }
"sizeof" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return SIZEOF; }
"break" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return BREAK; }
"continue" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return
CONTINUE; }
"goto" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return GOTO; }
"switch" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return SWITCH; }
"case" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return CASE; }
"default" { add_symbol(yytext,"NULL","Keyword",NULL,0,NULL,0,nestLevel,lineNo); return DEFAULT;
}
```

```
("\"")[^\n\"]*("\"") { addConstant(yytext,"String"); return STR_CONST; }
("\"")[^\n\"]* {
printf("Line No. %d ERROR: UNCLOSED STRING-%s\n", yylineno, yytext);
return 0;
}
("\'")({escape_sequences}|.)("\'") { addConstant(yytext,"Char"); return CHAR_CONST; }
("\'")((("\\")[^0abfnrtv\\\"\'][^\n\']*))|[^\n\'][^\n\']+)("\'") {
printf("Line No. %d ERROR: NOT A CHARACTER-%s\n", yylineno, yytext);
return 0;
}
[+-]?{num}*[.]{num}+ { addConstant(yytext, "Float"); return FLOAT_CONST; }
[+-]?{num}*[.]?{num}+[eE][+-]?{num}*[.]?{num}+ { addConstant(yytext, "Float"); return
FLOAT_CONST; }
{num}+ { addConstant(yytext, "Integer"); yylval = atoi(yytext); return INT_CONST; }
(_|{alpha})({alphanum}|_)* { strcpy(currIdenName,yytext); yylval = strdup(yytext); return ID; }
(_|{alpha})({alphanum}|_)*/\[ { strcpy(currIdenName,yytext); return ID; }
{ws} {}
. {
flag = 1;
if(yytext[0] == '#')
printf("Line No. %d PREPROCESSOR ERROR-%s\n", yylineno, yytext);
else
printf("Line No. %d ERROR ILLEGAL CHARACTER-%s\n", yylineno, yytext);
return 0;
}
%%
```

```
void addConstant(char* tok, char* type) {
if(constant_head == NULL) {
    token* new_node = (token*)malloc(sizeof(token));
    new_node->next = NULL;
    new_node->value = strdup(type);
    new_node->key = strdup(tok);
    constant_head = new_node;
    return;
}
token *cur = constant_head;
token *prev = NULL;
while(cur != NULL) {
    if(strcmp(cur->key, tok) == 0) return;
    prev = cur;
    cur = cur->next;
}
token* new_node = (token*)malloc(sizeof(token));
new_node->next = NULL;
new_node->value = strdup(type);
new_node->key = strdup(tok);
prev->next = new_node;
}


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

YACC Code

```
%{
 #include "stdio.h"
 #include "stdlib.h"
 #include "ctype.h"
 #include "string.h"

typedef struct node {
    char *key;
    char *value;
    struct node *next;
} token;

extern int lineNo;
extern int nestLevel;
extern char currData[100];
extern char currIdenName[100];

 extern int flag;
 extern token *constant_head;

 char funcName[100];

typedef struct symbol {
    char *name;                // Symbol name
    char *type;                // Data type
    char *class;               // Class (e.g., variable,function)
    int *boundaries_list;       // Array of boundaries
    int array_dimensions;       // Number of dimensions forarrays
    char **parameters_list;     // List of parameters for functions
    int procedure_definition;  // Flag: 1 if procedure, 0 otherwise
    int nesting_level;          // Scope level
    int line_declared;          // Line number where declared
    struct symbol *next;
} symbol;
```

```
symbol *symbol_table = NULL;
char** list_param=NULL;

void add_symbol(char* name, char* type, char* class, int* boundaries, int arr_dim, char**
params, int proc_def, int nest_level, int line_no);

void print_symbol_table(void);

 int index_=0;
 int arr_dim = 1;
 int yyerror();
 int yylex();

 %}
 %token INT DATA_TYPE SIZE_MODIFIER SIGN SCOPE STRUCT_UNION
 %token RETURN MAIN
 %token WHILE FOR DO
 %token BREAK CONTINUE GOTO
 %token ENDIF
 %token SWITCH CASE DEFAULT
 %token IF ELSE
 %token ID
 %token INT_CONST STR_CONST FLOAT_CONST CHAR_CONST
 %right OP_EQUAL
 %right '='
%left LOG
 %left '^'
 %left REL
 %left '+' '-'
 %left '*' '/' '%'
 %right SIZEOF
 %right NOT
 %left INC_DEC
 %expect 3
```

```
%expect 3
%start S
%%
S: content ;
content: declaration content | ;
declaration: variable_declaration
| function_declaration
| struct_dec ;
struct_dec: STRUCT_UNION ID '{' struct_body '}' ';'{

add_symbol((char*)$2, "struct", "struct", NULL, 0, NULL, 0, nestLevel, lineNo);
};
struct_body: variable_declaration struct_body
| ;
variable_declaration: datatype var ';'
| struct_initialise ;
struct_initialise: STRUCT_UNION ID var {add_symbol((char*)$2, "struct", "Variable", NULL, 0,
NULL, 0, nestLevel, lineNo);}
|STRUCT_UNION ID star var {add_symbol((char*)$2, "struct pointer", "Variable", NULL, 0, NULL,
0, nestLevel, lineNo);};
var: id_name
| id_name ',' var ;
id_name: ID arr_id |ID extended_id {add_symbol(currIdenName, currData, "Variable", NULL, 0,
NULL, 0, nestLevel, lineNo);};
extended_id:
'=' exp
|;

arr_id: '[' INT_CONST ']' initialise {

   arr_dim *= yylval;
   add_symbol(currIdenName, "Array", "Variable", NULL, arr_dim, NULL, 0, nestLevel, lineNo);
   arr_dim = 1;
}
```

```
| '[' INT_CONST ']' arr_id {
    arr_dim *= yylval;
    add_symbol(currIdenName, "Array", "Variable", NULL, arr_dim, NULL, 0, nestLevel, lineNo);
};


initialise: string_initialise
 | arr_initialise
 | ;
 string_initialise: '=' STR_CONST;
 arr_initialise: '=' multi_dim;
 multi_dim: '{' arr_elements '}';
 arr_elements: arr_values
 | '{' arr_values '}'
 | multi_dim
 | multi_dim ',' '{' arr_elements '}'
 | '{' arr_values '}' ',' arr_elements ;
 arr_values: constant multiple_arr_values;
 multiple_arr_values: ',' arr_values
 | ;

 datatype: INT
 | DATA_TYPE
 | SIZE_MODIFIER grammar
 | SIGN sign_grammar
 | SCOPE scope_grammar
 | datatype star;

 star: '*' star| '*';
 scope_grammar: INT| DATA_TYPE| SIGN grammar| ;
 sign_grammar: INT| SIZE_MODIFIER grammar| ;
 grammar: INT| ;
```

```
 function_declaration: fun_datatype fun_param;

 fun_datatype: datatype ID '('{strcpy(funcName,(char*)$2);};

 fun_param: param ')' statement {
 index_=0;
 add_symbol(funcName, "NULL", "Function", NULL, 0, list_param, 1, nestLevel, lineNo);
 list_param = (char**)malloc(sizeof(char*) * 100);
 }; //add to table //reset list

param: datatype all_param_id {
*(list_param+index_) = strdup(currData);
index_++;
}
| ;

 all_param_id: param_id multiple_param;

 multiple_param: ',' param| ;

 param_id: ID ext_param;

 ext_param: '[' ']'| ;
```

```
statement: expression_statement
| multiple_statements
| conditional_statement
| iterartive_statement
| return_statement
| break_statement
| continue_statement
| switch_statement
| variable_declaration;
multiple_statements: '{' block '}' ;
block: statement block
| ;
expression_statement: exp ';'| exp ',' expression_statement
| ';' ;
conditional_statement: IF '(' simple_exp ')' statement
extended_conditional_statement;
extended_conditional_statement: ELSE statement
| ;
iterartive_statement: WHILE '(' simple_exp ')' statement
| FOR '(' for_initialise simple_exp ';' exp ')'
| DO statement WHILE '(' simple_exp ')' ';';
switch_statement: SWITCH '('simple_exp')' '{' case_st DEFAULT ':'
statement '}';
case_st: CASE int_char_const ':' statement BREAK ';' case_st
| ;
```

```
int_char_const: INT_CONST
 | CHAR_CONST;
 for_initialise: variable_declaration
 | exp ';'
 | ';' ;
 return_statement: RETURN return_suffix;
 return_suffix: ';'
 | exp ';' ;
 break_statement: BREAK ';' ;
 continue_statement: CONTINUE ';' ;
 exp: identifier expression
 | simple_exp ;
 expression: '=' exp
 | OP_EQUAL exp
 | INC_DEC ;
 simple_exp: unary_relation_exp rel_exp_breakup;
 rel_exp_breakup: LOG unary_relation_exp rel_exp_breakup
 | ;
 unary_relation_exp: NOT unary_relation_exp
 | regular_exp ;
 regular_exp: arithmetic_exp regular_exp_breakup;
 regular_exp_breakup: REL arithmetic_exp
 | ;
 arithmetic_exp: arithmetic_exp operators factor
 | factor ;
```

```
 operators: '+'
 | '-'
 | '*'
| '/'
 |'^'
 | '%';
 factor: fun
 | identifier ;
 identifier: ID
 | '&' ID
 | identifier ext_identifier;
 ext_identifier: '[' exp ']'
 | '.' ID;
 | "->" ID;
 fun: '(' exp ')'
 | fun_call
 | constant;
 fun_call: ID '(' arg ')';
 arg: arg_list
 | ;
 arg_list: exp ext_arg;
 ext_arg: ',' exp ext_arg
 | ;
 constant: INT_CONST
 | STR_CONST
 | FLOAT_CONST
 | CHAR_CONST;
 %%
```

```
   extern FILE *yyin;
   extern int yylineno;
   extern char *yytext;

  int main() {
      list_param = (char**)malloc(sizeof(char*) * 100);
      yyin = fopen("input.c", "r");
      yyparse();

      if (flag == 0) {
          printf("VALID PARSE\n");

          // Printing the constant table
          token *cur = constant_head;
          printf("CONSTANT TABLE\n");
          printf("%-20s %-20s\n", "     CONSTANT", "      TYPE");
          printf("%-20s %-20s\n", "--------------------", "--------------------");
          while (cur != NULL) {
              printf("%-20s %-20s\n", cur->key, cur->value);
              printf("%-20s %-20s\n", "--------------------", "--------------------");
              cur = cur->next;
          }

          // Print the symbol table
          print_symbol_table();
      }
  }
```

```
void print_symbol_table() {
    symbol *current = symbol_table;
    printf("\nSYMBOL TABLE\n");
    printf("%-15s %-10s %-10s %-15s %-15s %-20s %-10s %-10s\n",
           "Name", "Type", "Class", "Array Dims", "Parameters", "Procedure", "Nesting", "Line
No.");
    printf("-------------------------------------------------------------------------------------
------------\n");

    while (current != NULL) {
        // Print symbol details
        printf("%-15s %-10s %-10s ", current->name, current->type, current->class);

        // Print array dimensions
        if (current->array_dimensions > 0) {
            printf("%-15d ", current->array_dimensions);
        } else {
            printf("%-15s ", "-");
        }

        // Print function parameters
        if (current->parameters_list != NULL) {
            printf("[");
            for (int i = 0; current->parameters_list[i] != NULL; i++) {
                printf("%s", current->parameters_list[i]);
                if (current->parameters_list[i + 1] != NULL) {
                    printf(", ");
```

```
                }
            }
            printf("] ");
        } else {
            printf("%-15s ", "-");
        }

        // Procedure definition flag
        printf("%-20s ", current->procedure_definition ? "Yes" : "No");

        // Nesting level and line number
        printf("%-10d %-10d\n", current->nesting_level, current->line_declared);

        current = current->next;
    }
}

 int yyerror(char *s){
 printf("Line No. : %d %s %s\n",yylineno, s, yytext);
 flag=1;
 printf("INVALID PARSE\n");

 }
```

```
void add_symbol(char* name, char* type, char* class, int* boundaries, int arr_dim,
                char** params, int proc_def, int nest_level, int line_no) {
    symbol *new_symbol = (symbol *)malloc(sizeof(symbol));
    new_symbol->name = strdup(name);
     new_symbol->type = NULL;
    if(type!=NULL)
    new_symbol->type = strdup(type);
    if(class != NULL)
    new_symbol->class = strdup(class);
    new_symbol->array_dimensions = arr_dim;
    new_symbol->parameters_list = params;
    new_symbol->procedure_definition = proc_def;
    new_symbol->nesting_level = nest_level;
    new_symbol->line_declared = line_no;

    // Add boundaries list if applicable
    if(boundaries != NULL) {
        new_symbol->boundaries_list = (int *)malloc(arr_dim * sizeof(int));
        for(int i = 0; i < arr_dim; i++) {
            new_symbol->boundaries_list[i] = boundaries[i];
        }
    }

    // Add to the symbol table
    new_symbol->next = symbol_table;
    symbol_table = new_symbol;
}
```

The First and Follow sets for each non-terminal:

First Sets:

First(S) = **First**(content) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, ε}
First(content) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, ε}
First(declaration) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE}
First(struct_dec) = {STRUCT_UNION}
First(struct_body) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, ε}
First(variable_declaration) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, STRUCT_UNION}
First(struct_initialise) = {STRUCT_UNION}
First(var) = {ID}
First(id_name) = {ID}
First(extended_id) = {[, =, ε}
First(arr_id) = {[}
First(initialise) = {=, ε}
First(string_initialise) = {=}
First(arr_initialise) = {=}
First(multi_dim) = {{}}
First(arr_elements) = {INT_CONST, FLOAT_CONST, CHAR_CONST, STR_CONST, {}}
First(arr_values) = {INT_CONST, FLOAT_CONST, CHAR_CONST, STR_CONST}
First(multiple_arr_values) = {,, ε}
First(datatype) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE}
First(star) = {*}
First(scope_grammar) = {INT, DATA_TYPE, SIGN, ε}
First(sign_grammar) = {INT, SIZE_MODIFIER, ε}
First(grammar) = {INT, ε}
First(function_declaration) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE}
First(fun_datatype) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE}
First(fun_param) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE,)}
First(param) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, ε}
First(all_param_id) = {ID}
First(multiple_param) = {,, ε}
First(param_id) = {ID}
First(ext_param) = {[, ε}

First(statement) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST, ;, {, IF, WHILE, FOR, DO, SWITCH, RETURN, BREAK, CONTINUE, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, STRUCT_UNION}
First(multiple_statements) = {{}}
First(block) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST, ;, {, IF, WHILE, FOR, DO, SWITCH, RETURN, BREAK, CONTINUE, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, STRUCT_UNION, ε}
First(expression_statement) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST, ;}
First(conditional_statement) = {IF}
First(extended_conditional_statement) = {ELSE, ε}
First(iterative_statement) = {WHILE, FOR, DO}
First(switch_statement) = {SWITCH}
First(case_st) = {CASE, ε}
First(int_char_const) = {INT_CONST, CHAR_CONST}
First(for_initialise) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, STRUCT_UNION, ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST, ;}
First(return_statement) = {RETURN}
First(return_suffix) = {;, ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
First(break_statement) = {BREAK}
First(continue_statement) = {CONTINUE}
First(exp) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
First(expression) = {=, OP_EQUAL, INC_DEC}
First(simple_exp) = {NOT, ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
First(rel_exp_breakup) = {LOG, ε}
First(unary_relation_exp) = {NOT, ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
First(regular_exp) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
First(regular_exp_breakup) = {REL, ε}
First(arithmetic_exp) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
First(operators) = {+, -, *, /, ^, %}
First(factor) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
First(identifier) = {ID, &}
First(ext_identifier) = {[, ., ->, ε}
First(fun) = {(, ID, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
First(fun_call) = {ID}
First(arg) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST, ε}
First(arg_list) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
First(ext_arg) = {,, ε}

First(constant) = {INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}

Follow Sets

Follow(S) = {$}
Follow(content) = {$}
Follow(declaration) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }}
Follow(struct_dec) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }}
Follow(struct_body) = {}}
Follow(variable_declaration) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, },), CASE, DEFAULT}
Follow(struct_initialise) = {;}
Follow(var) = {;}
Follow(id_name) = {,, ;}
Follow(extended_id) = {,, ;}
Follow(arr_id) = {,, ;, =}
Follow(initialise) = {,, ;}
Follow(string_initialise) = {,, ;}
Follow(arr_initialise) = {,, ;}
Follow(multi_dim) = {,, ;}
Follow(arr_elements) = {}}
Follow(arr_values) = {,, }}
Follow(multiple_arr_values) = {,, }}
Follow(datatype) = {ID, *}
Follow(star) = {ID, *}
Follow(scope_grammar) = {ID, *}
Follow(sign_grammar) = {ID, *}
Follow(grammar) = {ID, *}
Follow(function_declaration) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }}
Follow(fun_datatype) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE,)}
Follow(fun_param) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }}
Follow(param) = {)}
Follow(all_param_id) = {)}
Follow(multiple_param) = {)}
Follow(param_id) = {,,)}
Follow(ext_param) = {,,)}

Follow(statement) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }, ELSE, CASE, DEFAULT, BREAK, WHILE}
Follow(multiple_statements) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }, ELSE, CASE, DEFAULT, BREAK, WHILE}
Follow(block) = {}}
Follow(expression_statement) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }, ELSE, CASE, DEFAULT, BREAK, WHILE}
Follow(conditional_statement) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }, ELSE, CASE, DEFAULT, BREAK, WHILE}
Follow(extended_conditional_statement) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }, ELSE, CASE, DEFAULT, BREAK, WHILE}
Follow(iterative_statement) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }, ELSE, CASE, DEFAULT, BREAK, WHILE}
Follow(switch_statement) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }, ELSE, CASE, DEFAULT, BREAK, WHILE}
Follow(case_st) = {DEFAULT}
Follow(int_char_const) = {:}
Follow(for_initialise) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST, NOT}
Follow(return_statement) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }, ELSE, CASE, DEFAULT, BREAK, WHILE}
Follow(return_suffix) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }, ELSE, CASE, DEFAULT, BREAK, WHILE}
Follow(break_statement) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }, ELSE, CASE, DEFAULT, BREAK, WHILE}
Follow(continue_statement) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }, ELSE, CASE, DEFAULT, BREAK, WHILE}
Follow(exp) = {;, ,,),], }}
Follow(expression) = {;, ,}
Follow(simple_exp) = {;, ,,),], }}
Follow(rel_exp_breakup) = {;, ,,),], }}
Follow(unary_relation_exp) = {LOG, ;, ,,),], }}
Follow(regular_exp) = {REL, LOG, ;, ,,),], }}
Follow(regular_exp_breakup) = {REL, LOG, ;, ,,),], }}
Follow(arithmetic_exp) = {REL, LOG, ;, ,,),], }, +, -, *, /, ^, %}
Follow(operators) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
Follow(factor) = {REL, LOG, ;, ,,),], }, +, -, *, /, ^, %}
Follow(identifier) = {=, OP_EQUAL, INC_DEC, REL, LOG, ;, ,,),], }, +, -, *, /, ^, %}
Follow(ext_identifier) = {=, OP_EQUAL, INC_DEC, REL, LOG, ;, ,,),], }, +, -, *, /, ^, %, [, ., ->}
Follow(fun) = {REL, LOG, ;, ,,),], }, +, -, *, /, ^, %}
Follow(fun_call) = {REL, LOG, ;, ,,),], }, +, -, *, /, ^, %}
Follow(arg) = {)}

Follow(arg_list) = {)}
Follow(ext_arg) = {)}
Follow(constant) = {REL, LOG, ;, ,,),], }, +, -, *, /, ^, %}

The Lookahead Set for each production

For each production, the Lookahead set is the union of the First set of the right-hand side and the Follow set of the left-hand side non-terminal if the right-hand side can derive ε.

Here is the updated version with only the word "Lookahead" bolded:

1. S -> content
   Lookahead = First(content) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $}
2. content -> declaration content
   Lookahead = First(declaration) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE}
3. content -> ε
   Lookahead = Follow(content) = {$}
4. declaration -> variable_declaration
   Lookahead = First(variable_declaration) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, STRUCT_UNION}
5. declaration -> function_declaration
   Lookahead = First(function_declaration) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE}
6. declaration -> struct_dec
   Lookahead = First(struct_dec) = {STRUCT_UNION}
7. struct_dec -> STRUCT_UNION ID '{' struct_body '}' ';'
   Lookahead = {STRUCT_UNION}
8. struct_body -> variable_declaration struct_body
   Lookahead = First(variable_declaration) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, STRUCT_UNION}
9. struct_body -> ε
   Lookahead = Follow(struct_body) = {}}
10. variable_declaration -> datatype var ';'
    Lookahead = First(datatype) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE}
11. variable_declaration -> struct_initialise
    Lookahead = First(struct_initialise) = {STRUCT_UNION}
12. struct_initialise -> STRUCT_UNION ID var
    Lookahead = {STRUCT_UNION}

13. struct_initialise -> STRUCT_UNION ID star var
    Lookahead = {STRUCT_UNION}
14. var -> id_name
    Lookahead = First(id_name) = {ID}
15. var -> id_name ',' var
    Lookahead = First(id_name) = {ID}
16. id_name -> ID extended_id
    Lookahead = {ID}
17. extended_id -> arr_id
    Lookahead = First(arr_id) = {[}
18. extended_id -> '=' exp
    Lookahead = {=}
19. extended_id -> ε
    Lookahead = Follow(extended_id) = {,, ;}
20. arr_id -> '[' INT_CONST ']' initialise
    Lookahead = {[}
21. arr_id -> '[' INT_CONST ']' arr_id
    Lookahead = {[}
22. initialise -> string_initialise
    Lookahead = First(string_initialise) = {=}
23. initialise -> arr_initialise
    Lookahead = First(arr_initialise) = {=}
24. initialise -> ε
    Lookahead = Follow(initialise) = {,, ;}
25. string_initialise -> '=' STR_CONST
    Lookahead = {=}
26. arr_initialise -> '=' multi_dim
    Lookahead = {=}
27. multi_dim -> '{' arr_elements '}'
    Lookahead = {{}
28. arr_elements -> arr_values
    Lookahead = First(arr_values) = {INT_CONST, FLOAT_CONST, CHAR_CONST, STR_CONST}
29. arr_elements -> '{' arr_values '}'
    Lookahead = {{}
30. arr_elements -> multi_dim
    Lookahead = First(multi_dim) = {{}
31. arr_elements -> multi_dim ',' '{' arr_elements '}'
    Lookahead = First(multi_dim) = {{}
32. arr_elements -> '{' arr_values '}' ',' arr_elements
    Lookahead = {{}
33. arr_values -> constant multiple_arr_values
    Lookahead = First(constant) = {INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}

34. multiple_arr_values -> ',' arr_values
    Lookahead = {,}
35. multiple_arr_values -> ε
    Lookahead = Follow(multiple_arr_values) = {,, }}
36. datatype -> INT
    Lookahead = {INT}
37. datatype -> DATA_TYPE
    Lookahead = {DATA_TYPE}
38. datatype -> SIZE_MODIFIER grammar
    Lookahead = {SIZE_MODIFIER}
39. datatype -> SIGN sign_grammar
    Lookahead = {SIGN}
40. datatype -> SCOPE scope_grammar
    Lookahead = {SCOPE}
41. datatype -> datatype star
    Lookahead = First(datatype) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE}
42. star -> '' *star*
    ***Lookahead** = {}*
43. star -> ''
    ***Lookahead** = {}*
44. scope_grammar -> INT
    Lookahead = {INT}
45. scope_grammar -> DATA_TYPE
    Lookahead = {DATA_TYPE}
46. scope_grammar -> SIGN grammar
    Lookahead = {SIGN}
47. scope_grammar -> ε
    Lookahead = Follow(scope_grammar) = {ID, *}
48. sign_grammar -> INT
    Lookahead = {INT}
49. sign_grammar -> SIZE_MODIFIER grammar
    Lookahead = {SIZE_MODIFIER}
50. sign_grammar -> ε
    Lookahead = Follow(sign_grammar) = {ID, *}
51. grammar -> INT
    Lookahead = {INT}
52. grammar -> ε
    Lookahead = Follow(grammar) = {ID, *}
53. function_declaration -> fun_datatype fun_param
    Lookahead = First(fun_datatype) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE}

54. fun_datatype -> datatype ID '('
    Lookahead = First(datatype) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE}
55. fun_param -> param ')' statement
    Lookahead = First(param) ∪ {)} = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE,)}
56. param -> datatype all_param_id
    Lookahead = First(datatype) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE}
57. param -> ε
    Lookahead = Follow(param) = {)}
58. all_param_id -> param_id multiple_param
    Lookahead = First(param_id) = {ID}
59. multiple_param -> ',' param
    Lookahead = {,}
60. multiple_param -> ε
    Lookahead = Follow(multiple_param) = {)}
61. param_id -> ID ext_param
    Lookahead = {ID}
62. ext_param -> '[' ']'
    Lookahead = {[}
63. ext_param -> ε
    Lookahead = Follow(ext_param) = {,,)}
64. statement -> expression_statement
    Lookahead = First(expression_statement) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST, ;}
65. statement -> multiple_statements
    Lookahead = First(multiple_statements) = {{}
66. statement -> conditional_statement
    Lookahead = First(conditional_statement) = {IF}
67. statement -> iterative_statement
    Lookahead = First(iterative_statement) = {WHILE, FOR, DO}
68. statement -> return_statement
    Lookahead = First(return_statement) = {RETURN}
69. statement -> break_statement
    Lookahead = First(break_statement) = {BREAK}
70. statement -> continue_statement
    Lookahead = First(continue_statement) = {CONTINUE}
71. statement -> switch_statement
    Lookahead = First(switch_statement) = {SWITCH}
72. statement -> variable_declaration
    Lookahead = First(variable_declaration) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, STRUCT_UNION}

73. multiple_statements -> '{' block '}'
    Lookahead = {{}
74. block -> statement block
    Lookahead = First(statement) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST, ;, {, IF, WHILE, FOR, DO, SWITCH, RETURN, BREAK, CONTINUE, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, STRUCT_UNION}
75. block -> ε
    Lookahead = Follow(block) = {}}
76. expression_statement -> exp ';'
    Lookahead = First(exp) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
77. expression_statement -> exp ',' expression_statement
    Lookahead = First(exp) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
78. expression_statement -> ';'
    Lookahead = {;}
79. conditional_statement -> IF '(' simple_exp ')' statement extended_conditional_statement
    Lookahead = {IF}
80. extended_conditional_statement -> ELSE statement
    Lookahead = {ELSE}
81. extended_conditional_statement -> ε
    Lookahead = Follow(extended_conditional_statement) = {STRUCT_UNION, INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, $, }, ELSE, CASE, DEFAULT}
82. iterative_statement -> WHILE '(' simple_exp ')' statement
    Lookahead = {WHILE}
83. iterative_statement -> FOR '(' expression_statement expression_statement ')' statement
    Lookahead = {FOR}
84. iterative_statement -> DO statement WHILE '(' simple_exp ')' ';'
    Lookahead = {DO}
85. switch_statement -> SWITCH '(' simple_exp ')' '{' case_st DEFAULT ':' statement '}'
    Lookahead = {SWITCH}
86. case_st -> CASE int_char_const ':' statement BREAK ';' case_st
    Lookahead = {CASE}
87. case_st -> ε
    Lookahead = Follow(case_st) = {DEFAULT}
88. int_char_const -> INT_CONST
    Lookahead = {INT_CONST}
89. int_char_const -> CHAR_CONST
    Lookahead = {CHAR_CONST}

90. for_initialise -> variable_declaration
    Lookahead = First(variable_declaration) = {INT, DATA_TYPE, SIZE_MODIFIER, SIGN, SCOPE, STRUCT_UNION}
91. for_initialise -> exp ';'
    Lookahead = First(exp) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
92. for_initialise -> ';'
    Lookahead = {;}
93. return_statement -> RETURN expression_statement
    Lookahead = {RETURN}
94. return_suffix -> ';'
    Lookahead = {;}
95. return_suffix -> exp ';'
    Lookahead = First(exp) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
96. break_statement -> BREAK ';'
    Lookahead = {BREAK}
97. continue_statement -> CONTINUE ';'
    Lookahead = {CONTINUE}
98. exp -> identifier expression
    Lookahead = First(identifier) = {ID, &}
99. exp -> simple_exp
    Lookahead = First(simple_exp) = {NOT, ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
100. expression -> '=' exp
    Lookahead = {=} 10
101. expression -> OP_EQUAL exp
    Lookahead = {OP_EQUAL}
102. expression -> INC_DEC
    Lookahead = {INC_DEC}
103. simple_exp -> unary_relation_exp rel_exp_breakup
    Lookahead = First(unary_relation_exp) = {NOT, ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
104. rel_exp_breakup -> LOG unary_relation_exp rel_exp_breakup
    Lookahead = {LOG}
105. rel_exp_breakup -> ε
    Lookahead = Follow(rel_exp_breakup) = {;, ,,),], }}
106. unary_relation_exp -> NOT unary_relation_exp
    Lookahead = {NOT}
107. unary_relation_exp -> regular_exp
    Lookahead = First(regular_exp) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}

108. regular_exp -> arithmetic_exp regular_exp_breakup
Lookahead = First(arithmetic_exp) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
109. regular_exp_breakup -> REL arithmetic_exp
Lookahead = {REL}
110. regular_exp_breakup -> ε
Lookahead = Follow(regular_exp_breakup) = {REL, LOG, ;, ,,),], }}
111. arithmetic_exp -> arithmetic_exp operators factor
Lookahead = First(arithmetic_exp) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
112. arithmetic_exp -> factor
Lookahead = First(factor) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
113. operators -> '+'
Lookahead = {+}
114. operators -> '-'
Lookahead = {-}
115. operators -> '*'
Lookahead = {*}
116. operators -> '/'
Lookahead = {/}
117. operators -> '^'
Lookahead = {^}
118. operators -> '%'
Lookahead = {%}
119. factor -> fun
Lookahead = First(fun) = {(, ID, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
120. factor -> identifier
Lookahead = First(identifier) = {ID, &}
121. identifier -> ID
Lookahead = {ID}
122. identifier -> '&' ID
Lookahead = {&}
123. identifier -> identifier ext_identifier
Lookahead = First(identifier) = {ID, &}
124. ext_identifier -> '[' exp ']'
Lookahead = {[}
125. ext_identifier -> '.' ID
Lookahead = {.}
126. ext_identifier -> "->" ID
Lookahead = {->}
127. fun -> '(' exp ')'
Lookahead = {(}

128. fun -> fun_call
Lookahead = First(fun_call) = {ID}
129. fun -> constant
Lookahead = First(constant) = {INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
130. fun_call -> ID '(' arg ')'
Lookahead = {ID}
131. arg -> arg_list
Lookahead = First(arg_list) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
132. arg -> ε
Lookahead = Follow(arg) = {)}
133. arg_list -> exp ext_arg
Lookahead = First(exp) = {ID, &, (, INT_CONST, STR_CONST, FLOAT_CONST, CHAR_CONST}
134. ext_arg -> ',' exp ext_arg
Lookahead = {,}
135. ext_arg -> ε
Lookahead = Follow(ext_arg) = {)}
136. constant -> INT_CONST
Lookahead = {INT_CONST}
137. constant -> STR_CONST
Lookahead = {STR_CONST}
138. constant -> FLOAT_CONST
Lookahead = {FLOAT_CONST}
139. constant -> CHAR_CONST
Lookahead = {CHAR_CONST}

TEST CASES - VALID PARSES:

1. Structure declaration and initialization

```c
#include<abcd.h>
struct Person {
    char name[50];
    int age;
};


int main() {
    struct Person p1;
    p1.age = 25;
    strcpy(p1->name, "John");
}
```

OUTPUT

```
~/Compiler-Design/Phase 2$ ./a.out
VALID PARSE
```

SYMBOL TABLE AND CONSTANT TABLE

```
CONSTANT TABLE
     CONSTANT                TYPE
--------------------  --------------------
50                    Integer
--------------------  --------------------
25                    Integer
--------------------  --------------------
"John"                String
--------------------  --------------------
```

```
SYMBOL TABLE
Name            Type        Class       Array Dims      Parameters      Procedure               Nesting     Line No.
-------------------------------------------------------------------------------------------------------
main            NULL        Function    -               [] Yes                  0           12
}               NULL        Delimiter   -               -               No                  0           12
;               NULL        Delimiter   -               -               No                  1           11
)               NULL        Delimiter   -               -               No                  1           11
,               NULL        Delimiter   -               -               No                  1           11
->              NULL        Operator    -               -               No                  1           11
(               NULL        Delimiter   -               -               No                  1           11
;               NULL        Delimiter   -               -               No                  1           10
=               NULL        Operator    -               -               No                  1           10
.               NULL        Delimiter   -               -               No                  1           10
Person          struct      Variable    -               -               No                  1           9
p1              Int         Variable    -               -               No                  1           9
;               NULL        Delimiter   -               -               No                  1           9
struct          NULL        Keyword     -               -               No                  1           9
{               NULL        Delimiter   -               -               No                  1           8
)               NULL        Delimiter   -               -               No                  0           8
(               NULL        Delimiter   -               -               No                  0           8
int             NULL        Keyword     -               -               No                  0           8
Person          struct      struct      -               -               No                  0           5
;               NULL        Delimiter   -               -               No                  0           5
}               NULL        Delimiter   -               -               No                  0           5
age             Int         Variable    -               -               No                  1           4
;               NULL        Delimiter   -               -               No                  1           4
int             NULL        Keyword     -               -               No                  1           4
name            Array       Variable    50              -               No                  1           3
;               NULL        Delimiter   -               -               No                  1           3
]               NULL        Delimiter   -               -               No                  1           3
[               NULL        Delimiter   -               -               No                  1           3
char            NULL        Keyword     -               -               No                  1           3
{               NULL        Delimiter   -               -               No                  1           2
struct          NULL        Keyword     -               -               No                  0           2
```

2. Function declaration

```c
#include<abcd.h>

int func(int a, char b, float c){
    printf("Hi");
    return 0;
}

int add(int x, int y){
    return x+y;
}

int main() {
    int x=1;
}
```

OUTPUT

```
~/Compiler-Design/Phase 2$ ./a.out
VALID PARSE
```

SYMBOL TABLE AND CONSTANT TABLE

```
CONSTANT TABLE
     CONSTANT                TYPE
------------------  ------------------
"Hi"                String
------------------  ------------------
0                   Integer
------------------  ------------------
1                   Integer
------------------  ------------------
```

SYMBOL TABLE

Name	Type	Class	Array Dims	Parameters	Procedure	Nesting	Line No.
main	NULL	Function	-	[] Yes	0	14	
}	NULL	Delimiter	-	-	No	0	14
x	Int	Variable	-	-	No	1	13
;	NULL	Delimiter	-	-	No	1	13
=	NULL	Operator	-	-	No	1	13
int	NULL	Keyword	-	-	No	1	13
{	NULL	Delimiter	-	-	No	1	12
)	NULL	Delimiter	-	-	No	0	12
(	NULL	Delimiter	-	-	No	0	12
int	NULL	Keyword	-	-	No	0	12
add	NULL	Function	-	[Int, Int] Yes	0	10	
}	NULL	Delimiter	-	-	No	0	10
;	NULL	Delimiter	-	-	No	1	9
+	NULL	Operator	-	-	No	1	9
return	NULL	Keyword	-	-	No	1	9
{	NULL	Delimiter	-	-	No	1	8
)	NULL	Delimiter	-	-	No	0	8
int	NULL	Keyword	-	-	No	0	8
,	NULL	Delimiter	-	-	No	0	8
int	NULL	Keyword	-	-	No	0	8
(	NULL	Delimiter	-	-	No	0	8
int	NULL	Keyword	-	-	No	0	8
func	NULL	Function	-	[float, float, float] Yes	0	6	
}	NULL	Delimiter	-	-	No	0	6
;	NULL	Delimiter	-	-	No	1	5
return	NULL	Keyword	-	-	No	1	5
;	NULL	Delimiter	-	-	No	1	4
)	NULL	Delimiter	-	-	No	1	4
(	NULL	Delimiter	-	-	No	1	4
{	NULL	Delimiter	-	-	No	1	3
)	NULL	Delimiter	-	-	No	0	3
float	NULL	Keyword	-	-	No	0	3
,	NULL	Delimiter	-	-	No	0	3
char	NULL	Keyword	-	-	No	0	3
,	NULL	Delimiter	-	-	No	0	3
int	NULL	Keyword	-	-	No	0	3
(	NULL	Delimiter	-	-	No	0	3
int	NULL	Keyword	-	-	No	0	3

3. For loop and if else construct

```c
#include <abcd.h>

int main() {
    int x=20;
    for(int i=0;i<10;i++) printf("Hello World");

    if(x==30) printf("x=30");
    else printf("x!=30");
}
```

OUTPUT

```
~/Compiler-Design/Phase 2$ ./a.out
VALID PARSE
```

SYMBOL TABLE AND CONSTANT TABLE

CONSTANT TABLE

CONSTANT	TYPE
20	Integer
0	Integer
10	Integer
"Hello World"	String
30	Integer
"x=30"	String
"x!=30"	String

SYMBOL TABLE

Name	Type	Class	Array Dims	Parameters	Procedure	Nesting	Line No.
main	NULL	Function	-	[] Yes	0	9	
}	NULL	Delimiter	-	-	No	0	9
;	NULL	Delimiter	-	-	No	1	8
)	NULL	Delimiter	-	-	No	1	8
(	NULL	Delimiter	-	-	No	1	8
else	NULL	Keyword	-	-	No	1	8
;	NULL	Delimiter	-	-	No	1	7
)	NULL	Delimiter	-	-	No	1	7
(	NULL	Delimiter	-	-	No	1	7
)	NULL	Delimiter	-	-	No	1	7
==	NULL	Operator	-	-	No	1	7
(	NULL	Delimiter	-	-	No	1	7
if	NULL	Keyword	-	-	No	1	7
;	NULL	Delimiter	-	-	No	1	5
)	NULL	Delimiter	-	-	No	1	5
(	NULL	Delimiter	-	-	No	1	5
)	NULL	Delimiter	-	-	No	1	5
++	NULL	Operator	-	-	No	1	5
;	NULL	Delimiter	-	-	No	1	5
<	NULL	Operator	-	-	No	1	5
i	Int	Variable	-	-	No	1	5
;	NULL	Delimiter	-	-	No	1	5
=	NULL	Operator	-	-	No	1	5
int	NULL	Keyword	-	-	No	1	5
(	NULL	Delimiter	-	-	No	1	5
for	NULL	Keyword	-	-	No	1	5
x	Int	Variable	-	-	No	1	4
;	NULL	Delimiter	-	-	No	1	4
=	NULL	Operator	-	-	No	1	4
int	NULL	Keyword	-	-	No	1	4
{	NULL	Delimiter	-	-	No	1	3
)	NULL	Delimiter	-	-	No	0	3
(	NULL	Delimiter	-	-	No	0	3
int	NULL	Keyword	-	-	No	0	3

4. While loop and array (single and multidimensional) declaration

```c
#include <abcd.h>

int main() {
    int x[2][2]={{1,2},{3,4}};
    float y[5]={1.1,2.2,3.3,4.4,5.5};
    int i=0;
    while(i<10){
        printf("%d ",i);
        i++;
    }
}
```

OUTPUT

```
~/Compiler-Design/Phase 2$ ./a.out
VALID PARSE
```

SYMBOL TABLE AND CONSTANT TABLE

```
CONSTANT TABLE
     CONSTANT              TYPE
------------------  ------------------
2                   Integer
------------------  ------------------
1                   Integer
------------------  ------------------
3                   Integer
------------------  ------------------
4                   Integer
------------------  ------------------
5                   Integer
------------------  ------------------
1.1                 Float
------------------  ------------------
2.2                 Float
------------------  ------------------
3.3                 Float
------------------  ------------------
4.4                 Float
------------------  ------------------
5.5                 Float
------------------  ------------------
0                   Integer
------------------  ------------------
10                  Integer
------------------  ------------------
"%d "               String
------------------  ------------------
```

SYMBOL TABLE

Name	Type	Class	Array Dims	Parameters	Procedure	Nesting	Line No.
main	NULL	Function	-	[] Yes	0	11	
}	NULL	Delimiter	-	-	No	0	11
}	NULL	Delimiter	-	-	No	1	10
;	NULL	Delimiter	-	-	No	2	9
++	NULL	Operator	-	-	No	2	9
;	NULL	Delimiter	-	-	No	2	8
)	NULL	Delimiter	-	-	No	2	8
,	NULL	Delimiter	-	-	No	2	8
(	NULL	Delimiter	-	-	No	2	8
{	NULL	Delimiter	-	-	No	2	7
)	NULL	Delimiter	-	-	No	1	7
<	NULL	Operator	-	-	No	1	7
(	NULL	Delimiter	-	-	No	1	7
while	NULL	Keyword	-	-	No	1	7
i	Int	Variable	-	-	No	1	6
;	NULL	Delimiter	-	-	No	1	6
=	NULL	Operator	-	-	No	1	6
int	NULL	Keyword	-	-	No	1	6
;	NULL	Delimiter	-	-	No	1	5
y	Array	Variable	20	-	No	1	5
}	NULL	Delimiter	-	-	No	1	5
,	NULL	Delimiter	-	-	No	2	5
,	NULL	Delimiter	-	-	No	2	5
,	NULL	Delimiter	-	-	No	2	5
,	NULL	Delimiter	-	-	No	2	5
{	NULL	Delimiter	-	-	No	2	5
=	NULL	Operator	-	-	No	1	5
]	NULL	Delimiter	-	-	No	1	5
[	NULL	Delimiter	-	-	No	1	5
float	NULL	Keyword	-	-	No	1	5
;	NULL	Delimiter	-	-	No	1	4
x	Array	Variable	4	-	No	1	4
x	Array	Variable	4	-	No	1	4
}	NULL	Delimiter	-	-	No	1	4
}	NULL	Delimiter	-	-	No	2	4
,	NULL	Delimiter	-	-	No	3	4
{	NULL	Delimiter	-	-	No	3	4
,	NULL	Delimiter	-	-	No	2	4
}	NULL	Delimiter	-	-	No	2	4
,	NULL	Delimiter	-	-	No	3	4
{	NULL	Delimiter	-	-	No	3	4
{	NULL	Delimiter	-	-	No	2	4
=	NULL	Operator	-	-	No	1	4
]	NULL	Delimiter	-	-	No	1	4
[	NULL	Delimiter	-	-	No	1	4
]	NULL	Delimiter	-	-	No	1	4
[	NULL	Delimiter	-	-	No	1	4
int	NULL	Keyword	-	-	No	1	4
{	NULL	Delimiter	-	-	No	1	3
)	NULL	Delimiter	-	-	No	0	3
(	NULL	Delimiter	-	-	No	0	3
int	NULL	Keyword	-	-	No	0	3

5. switch case statement

```c
#include <stdio.h>
int main() {
    int day;
    switch (day) {
        case 1:
            printf("Monday\n");
            break;
        case 2:
            printf("Tuesday\n");
            break;
        default:
            printf("Invalid input\n");
    }
    return 0;
}
```

OUTPUT

```
~/Compiler-Design/Phase 2$ ./a.out
VALID PARSE
```

SYMBOL TABLE AND CONSTANT TABLE

```
CONSTANT TABLE
     CONSTANT                    TYPE
--------------------  --------------------
1                     Integer
--------------------  --------------------
"Monday\n"            String
--------------------  --------------------
2                     Integer
--------------------  --------------------
"Tuesday\n"           String
--------------------  --------------------
"Invalid input. Please enter a number between 1 and 7.\n" String
--------------------  --------------------
0                     Integer
--------------------  --------------------
```

```
SYMBOL TABLE
Name            Type        Class       Array Dims      Parameters      Procedure               Nesting     Line No.
-------------------------------------------------------------------------------------------------------------------
main            NULL        Function    -               [] Yes                  0           16
}               NULL        Delimiter   -               -               No                  0           16
;               NULL        Delimiter   -               -               No                  1           15
return          NULL        Keyword     -               -               No                  1           15
}               NULL        Delimiter   -               -               No                  1           14
;               NULL        Delimiter   -               -               No                  2           13
)               NULL        Delimiter   -               -               No                  2           13
(               NULL        Delimiter   -               -               No                  2           13
:               NULL        Delimiter   -               -               No                  2           12
default         NULL        Keyword     -               -               No                  2           12
;               NULL        Delimiter   -               -               No                  2           11
break           NULL        Keyword     -               -               No                  2           11
;               NULL        Delimiter   -               -               No                  2           10
)               NULL        Delimiter   -               -               No                  2           10
(               NULL        Delimiter   -               -               No                  2           10
:               NULL        Delimiter   -               -               No                  2           9
case            NULL        Keyword     -               -               No                  2           9
;               NULL        Delimiter   -               -               No                  2           8
break           NULL        Keyword     -               -               No                  2           8
;               NULL        Delimiter   -               -               No                  2           7
)               NULL        Delimiter   -               -               No                  2           7
(               NULL        Delimiter   -               -               No                  2           7
:               NULL        Delimiter   -               -               No                  2           6
case            NULL        Keyword     -               -               No                  2           6
{               NULL        Delimiter   -               -               No                  2           5
)               NULL        Delimiter   -               -               No                  1           5
(               NULL        Delimiter   -               -               No                  1           5
switch          NULL        Keyword     -               -               No                  1           5
day             Int         Variable    -               -               No                  1           4
;               NULL        Delimiter   -               -               No                  1           4
int             NULL        Keyword     -               -               No                  1           4
{               NULL        Delimiter   -               -               No                  1           3
)               NULL        Delimiter   -               -               No                  0           3
(               NULL        Delimiter   -               -               No                  0           3
int             NULL        Keyword     -               -               No                  0           3
```

TEST CASES - INVALID PARSES:

1. Missing semicolon

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

Output:

```
user@user-Lenovo-V15-G2-ITL-U
Line No. : 3 syntax error }
INVALID PARSE
```

2. Wrong array initialization

```
int main(){
    int a[2][3]={{1,2},1,2};
    return;
}
```

Output:

```
user@user-Lenovo-V15-G2-ITL-Ua:
Line No. : 2 syntax error 1
INVALID PARSE
```

3. Missing parentheses

```
int main(){
    printf("xyx";
    return 0;
}
```

Output:

```
Line No. : 2 syntax error ;
INVALID PARSE
```

4. Misspelt keywords

```
int main(){
    switch(s){
        case 1: a++;
                brake;
        default: b++;
    }
}
```

Output:

```
Line No. : 4 syntax error brake
INVALID PARSE
```

5. Wrong syntax of for loop

```
int main(){
    for(int i=0;i++){
        a+=9;
    }
    return 0;
}
```

Output:

```
Line No. : 2 syntax error ++
INVALID PARSE
```

6. Global use of for/if/while

```
//Global use of for/if
#include<avfrg.h>

for(int i=0;i<7;i++);

int main(){

}
```

Output:

```
~/Compiler-Design/Phase 2$ ./a.out
Line No. : 3 syntax error for
INVALID PARSE
```

7. Invalid Header Format

```c
//Invalid header format
#include<abcde>

int main(){
  printf("hello");
}
```

Output:

```
~/Compiler-Design/Phase 2$ ./a.out
Line No. 2 PREPROCESSOR ERROR-#
```