CS304- COMPILER DESIGN LAB

PARSER FOR C - LANGUAGE

Group Members:

Aahil Rafiq	**Anumola Yoga Anil Kumar**	**Vasa Taraka Ravindra**
221CS101	221CS110	221CS162

V SEMESTER B-TECH CSE- S1

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL

2024 – 2024

Abstract:

This project encompasses the design and implementation of both a lexical analyzer and a syntax analyzer for the C programming language, using LEX/Flex and YACC/Bison, respectively. The focus is on accurately processing C's lexical and syntactic components, ensuring correct tokenization, error handling, and parsing.

Phase 1: Lexical Analyzer (Scanner) The first phase involves creating a lexical analyzer that identifies various lexical components, including keywords, identifiers, constants, string literals, and comments. Special attention is given to handling nested comments and providing meaningful error messages for issues such as unclosed comments or strings. The scanner is designed based on C's grammar and lexical specifications, with a symbol table and constants table implemented using hash organization for efficient storage of identified tokens. The project report covers the Flex script development, symbol table design, and the implementation of scanner routines, along with a discussion of recognized tokens and their meanings. Test cases are presented to validate the scanner's performance, with a focus on error detection and handling.

Phase 2: Syntax Analyzer (Parser) The second phase extends the project by implementing a syntax analyzer using YACC/Bison. This parser processes the tokens generated by the scanner, adhering to C's grammar while minimizing shift-reduce conflicts and avoiding reduce-reduce conflicts. The parser is integrated with the scanner, updating the symbol and constants tables as it parses the input. The grammar is augmented to test its robustness, and the parser is tested on various C programs. The report details the parser's development, including the grammar rules, First and Follow sets, and the handling of shift-reduce conflicts. Test cases highlight the parser's effectiveness in correctly parsing valid code and identifying syntactic errors.

Together, these phases provide a comprehensive overview of building a C language compiler, from lexical analysis to syntax parsing. The project demonstrates the challenges and solutions involved in processing complex language constructs, ensuring accurate tokenization, and achieving reliable parsing, with robust error detection and reporting.

Contents:

Introduction

Syntax analysis, often referred to as parsing, is the process of analyzing a sequence of tokens generated by lexical analysis to determine its grammatical structure according to a given grammar. This phase is essential in compiling and interpreting programming languages, as it transforms the linear sequence of tokens into a structured representation, often a parse tree.

Syntax Analysis

Syntax analysis involves checking the sequence of tokens against the rules of a formal grammar to ensure that the token sequence adheres to the syntactical rules of the programming language or data format. It aims to verify the correctness of the token sequence and to create a structured representation of the code.

1. **Parser:**
    a. A parser is a program or module responsible for performing syntax analysis. It takes the tokens produced by the lexer and arranges them into a structure that reflects the grammatical rules of the language.
    b. Common types of parsers include recursive descent parsers, shift-reduce parsers, and LR parsers.
2. **Grammar:**

    The grammar defines the rules of the language in a formal way. It specifies how tokens can be combined to form valid statements, expressions, and other language constructs.

3. **Parse Tree :**
    a. The parse tree is a tree representation of the syntactic structure of the input token sequence, where each node represents a grammatical construct.
    b. Parse Tree is a simplified, more abstract representation that omits some of the syntax details and focuses on the logical structure of the code.

Steps in Syntax Analysis:

1. **Token Stream Input:**
    - The parser receives a stream of tokens generated by the lexer. Each token represents a syntactic unit like keywords, operators, identifiers, and literals.
2. **Parsing:**
    - The parser applies grammar rules to the token stream to build a parse tree or AST. It checks if the sequence of tokens conforms to the grammatical structure defined by the language.
3. **Error Detection:**
    - The parser identifies syntax errors in the token stream if the sequence does not match any valid grammar rule. Syntax errors are reported to the user for

correction.

4. **Generate Output:**
    - The output of syntax analysis is typically a parse tree that represents the syntactic structure of the input code. This structured representation is then used in subsequent phases of compilation or interpretation, such as semantic analysis and code generation.

Tools:

- **Yacc (Yet Another Compiler Compiler):**
    - Yacc is a tool used to generate parsers based on a given grammar. It produces C code for the parser that can be compiled and linked with the lexer.
- **Bison:**
    - Bison is an improved version of Yacc that is more widely used. It provides similar functionality with additional features and better error handling.

C Program

This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned in account. Finally, a file called lex.yy.c is generated, which when executed recognizes the tokens present in the C program which was given as an input.

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

Design of Programs

Code:

Lex code :

```
%{
        #include <stdio.h>
        #include <string.h>
        #include "y.tab.h"
        #include "lib/table.h"

        char current_identifier[20];
        char current_type[20];
        char current_value[20];
  char current_function[20];
        char previous_operator[20];
        int yylineno;
        int flag;

%}
```

```
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 MLCOMMENT
DE "define"
IN "include"

%%

            int nested_count = 0;
            int check_nested = 0;

\n          {yylineno++;}
"#include"[ ]*"<"{alpha}({alphanum})*".h>"   { }
"#define"[ ]+(_|{alpha})({alphanum})*[ ]*(.)+   { }
"//".*  { }

"/*"([^*]|\*+[^/*])*\*+"/" {}

"[" {return *yytext;}
"]" {return *yytext;}
"(" {return *yytext;}
")" {return *yytext;}
"{" {return *yytext;}
"}" {return *yytext;}
"," {return *yytext;}
";" {return *yytext;}



"char"              { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword"); return CHAR;}
"double"            { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword"); return DOUBLE;}
"else"              { insert_SymbolTable_line(yytext, yylineno); insert_SymbolTable(yytext, "Keyword");
return ELSE;}
"float"             { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");return FLOAT;}
"while"             { insert_SymbolTable(yytext, "Keyword"); return WHILE;}
"do"                { insert_SymbolTable(yytext, "Keyword"); return DO;}
"for"               { insert_SymbolTable(yytext, "Keyword"); return FOR;}
"if"                { insert_SymbolTable(yytext, "Keyword"); return IF;}
"int"               { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");return INT;}
"long"              { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return LONG;}
```

```
"return"{ insert_SymbolTable(yytext, "Keyword");  return RETURN;}
"short"             { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return SHORT;}
"signed"{ strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return SIGNED;}
"sizeof"  { insert_SymbolTable(yytext, "Keyword");  return SIZEOF;}
"struct"  { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return STRUCT;}
"unsigned"        { insert_SymbolTable(yytext, "Keyword");  return UNSIGNED;}
"void"              { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return VOID;}
"break"             { insert_SymbolTable(yytext, "Keyword");  return BREAK;}
"continue"        { insert_SymbolTable(yytext, "Keyword");  return CONTINUE;}
"goto"              { insert_SymbolTable(yytext, "Keyword");  return GOTO;}
"switch"{ insert_SymbolTable(yytext, "Keyword");  return SWITCH;}
"case"              { insert_SymbolTable(yytext, "Keyword");  return CASE;}
"default"           { insert_SymbolTable(yytext, "Keyword");  return DEFAULT;}

("\"")[^\n\"]*("\"")                  {strcpy(current_value,yytext); insert_ConstantTable(yytext,"String
Constant"); return string_constant;}
("\"")[^\n\"]*                        { printf("Line No. %d ERROR: UNCLOSED STRING - %s\n", yylineno, yytext);
return 0;}
("\'")(("\\"({escape_sequences}))|.)("\'")   {strcpy(current_value,yytext);
insert_ConstantTable(yytext,"Character Constant"); return character_constant;}
("\'")((((("\\")[^0abfnrtv\\\"\'][^\n\']*))|[^\n\''][^\n\'']+)("\'") {printf("Line No. %d ERROR: NOT A
CHARACTER - %s\n", yylineno, yytext); return 0; }
{num}+(\.{num}+)?e{num}+                    {strcpy(current_value,yytext); insert_ConstantTable(yytext,
"Floating Constant"); return float_constant;}
{num}+\.{num}+                      {strcpy(current_value,yytext); insert_ConstantTable(yytext, "Floating
Constant"); return float_constant;}
{num}+                              {strcpy(current_value,yytext); insert_ConstantTable(yytext, "Number
Constant"); return integer_constant;}
(_|{alpha})({alpha}|{alpha}|_)*
        {strcpy(current_identifier,yytext);insert_SymbolTable(yytext,"Identifier");  return identifier;}
(_|{alpha})({alpha}|{alpha}|_)*/\[
        {strcpy(current_identifier,yytext);insert_SymbolTable(yytext,"Array Identifier");  return identifier;}
{ws}                                                                                    ;

"+"     {return *yytext;}
"-"     {return *yytext;}
"*"     {return *yytext;}
"/"     {return *yytext;}
"="     {return *yytext;}
"%"     {return *yytext;}
"&"     {return *yytext; }
"^"     {return *yytext; }
"++" {return INCREMENT;}
```

```
"--" {return DECREMENT;}
"!"         {return NOT;}
"+=" {return ADD_EQUAL;}
"-=" {return SUBTRACT_EQUAL;}
"*=" {return MULTIPLY_EQUAL;}
"/=" {return DIVIDE_EQUAL;}
"%=" {return MOD_EQUAL;}
"&&" {return AND_AND;}

"||"      {return OR_OR;}
">"               {return GREAT;}
"<"               {return LESS;}
">="      {return GREAT_EQUAL;}
"<="      {return LESS_EQUAL;}
"=="      {return EQUAL;}
"!="      {return NOT_EQUAL;}
.                 { 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;}

%%
```

YACC :

```
%{
        void yyerror(char* s);
        int yylex();
        #include <stdio.h>
        #include <stdlib.h>
        #include <ctype.h>
        #include <string.h>
        #include "lib/table.h"
        void insert_type();
        void insert_value();
        void insert_dimensions();
        void insert_parameters();
        extern int flag;
        int insert_flag = 0;
```

```
        extern char current_identifier[20];
        extern char current_type[20];
        extern char current_value[20];
  extern char current_function[20];
        extern char previous_operator[20];

%}

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

%token identifier
%token integer_constant string_constant float_constant character_constant

%nonassoc ELSE

%right MOD_EQUAL
%right MULTIPLY_EQUAL DIVIDE_EQUAL
%right ADD_EQUAL SUBTRACT_EQUAL
%right '='

%left OR_OR
%left AND_AND
%left '^'
%left EQUAL NOT_EQUAL
%left LESS_EQUAL LESS GREAT_EQUAL GREAT
%left '+' '-'
%left '*' '/' '%'

%right SIZEOF
%right NOT
%left INCREMENT DECREMENT


%start begin_parse
```

```
%%
begin_parse
                        : declarations;

declarations
                        : declaration declarations
                        |
                        ;

declaration
                        : variable_dec
                        | function_dec
                        | structure_dec;

structure_dec
                        : STRUCT identifier { insert_type(); } '{' structure_content  '}' ';';

structure_content : variable_dec structure_content | ;

variable_dec
                        : datatype variables ';'
                        | structure_initialize;

structure_initialize
                        : STRUCT identifier variables;

variables
                        : identifier_name multiple_variables;

multiple_variables
                        : ',' variables
                        | ;

identifier_name
                        : identifier { insert_type(); } extended_identifier;

extended_identifier : array_identifier | '='{strcpy(previous_operator,"=");} expression ;

array_identifier
                        : '[' array_dims
                        | ;

array_dims
```

```
                    : integer_constant {insert_dimensions();} ']' initilization
                    | ']' string_initilization;

initilization
                    : string_initilization
                    | array_initialization
                    | ;

string_initilization
                    : '='{strcpy(previous_operator,"=");} string_constant { insert_value(); };

array_initialization
                    : '='{strcpy(previous_operator,"=");} '{' array_values '}';

array_values
                    : integer_constant multiple_array_values;

multiple_array_values
                    : ',' array_values
                    | ;


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

unsigned_grammar
                    : INT | LONG long_grammar | SHORT short_grammar | ;

signed_grammar
                    : INT | LONG long_grammar | SHORT short_grammar | ;

long_grammar
                    : INT | ;

short_grammar
                    : INT | ;

function_dec
```

```
                        : function_datatype function_parameters;

function_datatype
                        : datatype identifier '('  {strcpy(current_function,current_identifier);
insert_type();};

function_parameters
                        : parameters ')' statement;

parameters
                        : datatype all_parameter_identifiers | ;

all_parameter_identifiers
                        : parameter_identifier multiple_parameters;

multiple_parameters
                        : ',' parameters
                        | ;

parameter_identifier
                        : identifier { insert_parameters(); insert_type(); } extended_parameter;

extended_parameter
                        : '[' ']'
                        | ;

statement
                        : expression_statment | multiple_statement
                        | conditional_statements | iterative_statements
                        | return_statement | break_statement
                        | variable_dec;

multiple_statement
                        : '{' statments '}' ;

statments
                        : statement statments
                        | ;

expression_statment
                        : expression ';'
                        | ';' ;
```

```
conditional_statements
                        : IF '(' simple_expression ')' statement extended_conditional_statements;

extended_conditional_statements
                        : ELSE statement
                        | ;

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

for_initialization
                        : variable_dec
                        | expression ';'
                        | ';' ;

return_statement
                        : RETURN return_suffix;

return_suffix
                        : ';'
                        | expression ';' ;

break_statement
                        : BREAK ';' ;

expression
                        : iden expressions
                        | simple_expression ;

expressions
                        : '='{strcpy(previous_operator,"=");} expression
                        | ADD_EQUAL{strcpy(previous_operator,"+=");} expression
                        | SUBTRACT_EQUAL{strcpy(previous_operator,"-=");} expression
                        | MULTIPLY_EQUAL{strcpy(previous_operator,"*=");} expression
                        | DIVIDE_EQUAL{strcpy(previous_operator,"/=");} expression
                        | MOD_EQUAL{strcpy(previous_operator,"%=");} expression
                        | INCREMENT
                        | DECREMENT ;

simple_expression
                        : and_expression simple_expression_breakup;
```

```
simple_expression_breakup
                        : OR_OR and_expression simple_expression_breakup | ;

and_expression
                        : unary_relation_expression and_expression_breakup;

and_expression_breakup
                        : AND_AND unary_relation_expression and_expression_breakup
                        | ;

unary_relation_expression
                        : NOT unary_relation_expression
                        | regular_expression ;

regular_expression
                        : sum_expression regular_expression_breakup;

regular_expression_breakup
                        : relational_operators sum_expression
                        | ;

relational_operators
                        : GREAT_EQUAL{strcpy(previous_operator,">=");}
                        | LESS_EQUAL{strcpy(previous_operator,"<=");}
                        | GREAT{strcpy(previous_operator,">");}
                        | LESS{strcpy(previous_operator,"<");}
                        | EQUAL{strcpy(previous_operator,"==");}
                        | NOT_EQUAL{strcpy(previous_operator,"!=");} ;

sum_expression
                        : sum_expression sum_operators term
                        | term ;

sum_operators
                        : '+'
                        | '-' ;

term
                        : term multiply_operators factor
                        | factor ;

multiply_operators
```

```
                        : '*' | '/' | '%' ;

factor
                        : func | iden ;

iden
                        : identifier
                        | iden extended_iden;

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

func
                        : '('{strcpy(previous_operator,"(");} expression ')'
                        | func_call | constant;

func_call
                        : identifier '('{strcpy(previous_operator,"(");} arguments ')';

arguments
                        : arguments_list | ;

arguments_list
                        : expression extended_arguments;

extended_arguments
                        : ',' expression extended_arguments
                        | ;

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

%%

extern FILE *yyin;
extern int yylineno;
extern char *yytext;
void insert_SymbolTable_type(char *,char *);
void insert_SymbolTable_value(char *, char *);
```

```
void insert_ConstantTable(char *, char *);
void insert_SymbolTable_arraydim(char *, char *);
void insert_SymbolTable_funcparam(char *, char *);
void printSymbolTable();
void printConstantTable();


int main()
{
	yyparse();

	if(flag == 0)
	{
		printf("VALID PARSE\n");
		printf("%30s SYMBOL TABLE \n", " ");
		printf("%30s %s\n", " ", "------------");
		printSymbolTable();

		printf("\n\n%30s CONSTANT TABLE \n", " ");
		printf("%30s %s\n", " ", "--------------");
		printConstantTable();
	}
}

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

void insert_type()
{
	insert_SymbolTable_type(current_identifier,current_type);
}

void insert_value()
{
	if(strcmp(previous_operator, "=") == 0)
	{	insert_SymbolTable_value(current_identifier,current_value);
	}
}
```

```
void insert_dimensions()
{
   insert_SymbolTable_arraydim(current_identifier, current_value);
}

void insert_parameters()
{
   insert_SymbolTable_funcparam(current_function, current_identifier);
}

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

HASHTABLE :

```
// table.c

#include "table.h"
#include <stdio.h>
#include <string.h>

// Define the tables
struct ConstantTable CT[1000];
struct SymbolTable ST[1000];
extern int yylineno;

// Hash function to generate hash values for the strings
unsigned long hash(unsigned char *str) {
   unsigned long hash = 5381;
   int c;
   while ((c = *str++)) {
      hash = ((hash << 5) + hash) + c; // hash * 33 + c
   }
   return hash;
}

// Search in the Constant Table
int search_ConstantTable(char *str) {
   unsigned long temp_val = hash((unsigned char *)str);
   int val = temp_val % 1000;
```

```
    if (CT[val].exist == 0) {
        return 0;
    } else if (strcmp(CT[val].constant_name, str) == 0) {
        return 1;
    } else {
        for (int i = val + 1; i != val; i = (i + 1) % 1000) {
            if (strcmp(CT[i].constant_name, str) == 0) {
                return 1;
            }
        }
        return 0;
    }
}

// Insert into the Constant Table
void insert_ConstantTable(char *name, char *type) {
    if (search_ConstantTable(name)) {
        return;
    } else {
        unsigned long temp_val = hash((unsigned char *)name);
        int val = temp_val % 1000;

        if (CT[val].exist == 0) {
            strcpy(CT[val].constant_name, name);
            strcpy(CT[val].constant_type, type);
            CT[val].exist = 1;
            return;
        }

        for (int i = val + 1; i != val; i = (i + 1) % 1000) {
            if (CT[i].exist == 0) {
                strcpy(CT[i].constant_name, name);
                strcpy(CT[i].constant_type, type);
                CT[i].exist = 1;
                break;
            }
        }
    }
}

// Print the Constant Table
void printConstantTable() {
```

```
    printf("%20s | %20s\n", "CONSTANT", "TYPE");
    for (int i = 0; i < 1000; ++i) {
        if (CT[i].exist == 0)
            continue;

        printf("%20s | %20s\n", CT[i].constant_name, CT[i].constant_type);
    }
}

// Search in the Symbol Table
int search_SymbolTable(char *str) {
    unsigned long temp_val = hash((unsigned char *)str);
    int val = temp_val % 1000;

    if (ST[val].exist == 0) {
        return 0;
    } else if (strcmp(ST[val].symbol_name, str) == 0) {
        return 1;
    } else {
        for (int i = val + 1; i != val; i = (i + 1) % 1000) {
            if (strcmp(ST[i].symbol_name, str) == 0) {
                return 1;
            }
        }
        return 0;
    }
}

// Insert into the Symbol Table
void insert_SymbolTable(char *name, char *class) {
    if (search_SymbolTable(name)) {
        return;
    } else {
        unsigned long temp_val = hash((unsigned char *)name);
        int val = temp_val % 1000;

        if (ST[val].exist == 0) {
            strcpy(ST[val].symbol_name, name);
            strcpy(ST[val].class, class);
            ST[val].line_number = yylineno;
            ST[val].exist = 1;
            return;
        }
```

```
      for (int i = val + 1; i != val; i = (i + 1) % 1000) {
         if (ST[i].exist == 0) {
            strcpy(ST[i].symbol_name, name);
            strcpy(ST[i].class, class);
            ST[i].exist = 1;
            break;
         }
      }
   }
}

// Update Symbol Table entries
void insert_SymbolTable_type(char *str1, char *str2) {
   for (int i = 0; i < 1000; i++) {
      if (strcmp(ST[i].symbol_name, str1) == 0) {
         strcpy(ST[i].symbol_type, str2);
      }
   }
}

void insert_SymbolTable_value(char *str1, char *str2) {
   for (int i = 0; i < 1000; i++) {
      if (strcmp(ST[i].symbol_name, str1) == 0) {
         strcpy(ST[i].value, str2);
      }
   }
}

void insert_SymbolTable_arraydim(char *str1, char *dim) {
   for (int i = 0; i < 1000; i++) {
      if (strcmp(ST[i].symbol_name, str1) == 0) {
         strcpy(ST[i].array_dimensions, dim);
      }
   }
}

void insert_SymbolTable_funcparam(char *str1, char *param) {
   for (int i = 0; i < 1000; i++) {
      if (strcmp(ST[i].symbol_name, str1) == 0) {
         strcat(ST[i].parameters, " ");
         strcat(ST[i].parameters, param);
      }
```

```
    }
}

void insert_SymbolTable_line(char *str1, int line) {
    for (int i = 0; i < 1000; i++) {
        if (strcmp(ST[i].symbol_name, str1) == 0) {
            ST[i].line_number = line;
        }
    }
}

#include <stdio.h>

void printSeparator() {
    printf("+------------+--------------------+------------+------------+------------+------------+------------+\n");
}

void printSymbolTable() {
    printSeparator();
    printf("| %10s | %18s | %10s | %10s | %10s | %10s | %10s |\n",
           "SYMBOL", "CLASS", "TYPE", "VALUE", "DIMENSIONS", "PARAMETERS", "LINE NO");
    printSeparator();
    for (int i = 0; i < 1000; ++i) {
        if (ST[i].exist == 0)
            continue;
        printf("| %10s | %18s | %10s | %10s | %10s | %10s | %10d |\n",
               ST[i].symbol_name, ST[i].class, ST[i].symbol_type, ST[i].value,
               ST[i].array_dimensions, ST[i].parameters, ST[i].line_number);
    }
    printSeparator();
}
```

Explanation:

1. **YACC/Bison Script Structure**: The YACC script is divided into three sections: definitions, rules, and user code. The **definitions** section includes declarations and external variables. The **rules** section contains grammar rules and actions. The **user code** section is used for additional C code that interacts with the parser.
2. **Grammar Input**: The grammar must be entered in YACC format, which specifies how tokens (from the scanner) combine to form valid statements and constructs in C.
3. **Shift-Reduce Conflicts**: These occur when the parser has to choose between shifting (reading more input) or reducing (applying a grammar rule). Minimizing these conflicts involves careful grammar design.
4. **Reduce-Reduce Conflicts**: These occur when the parser can reduce by more than one rule. This should be avoided in your final grammar to ensure clear and unambiguous parsing.
5. **Symbol Table Implementation**: The symbol table should be implemented as a stack to manage scope and symbol information efficiently. Each entry includes fields such as symbol name, type, class, boundaries, dimensions, parameters, and nesting level.
6. **Integration with Scanner**: The parser interacts with the scanner to retrieve tokens and update symbol tables. Each token retrieved by the parser may cause updates to the symbol table or other data structures.

Test Cases:

Test Case	Output	Status
1	VALID PARSE	PASS
2	VALID PARSE	PASS
3	Line No. : 14 syntax error else INVALID PARSE	FAIL
4	VALID PARSE	PASS
5	Line No. : 3 syntax error { INVALID PARSE	FAIL

Results & Future work:

Parser correctly handles various C language constructs, generates accurate parse trees, and manages syntax errors effectively.

- Symbol table integration successfully manages scope and symbol information.

- Testing shows good coverage of typical C constructs and clear error messages with line numbers.
- Grammar has been extended to include additional features without issues.
- Future work includes refining grammar, enhancing error reporting, optimizing performance, and developing user interface tools.

References

1.https://www.geeksforgeeks.org/write-regular-expressions/
2.https://www.lysator.liu.se/c/ANSI-C-grammar-y.html