COMPILERS LAB REPORT

Assignment 2: Parser for Custom Language

Objective

The objective of this assignment is to develop a parser for the custom programming language with _RP keywords and 095 prefixed identifiers. The parser must:

- Handle all language components from **Assignment 1**.
- Use **unambiguous production rules** with defined precedence and associativity.
- Implement **error recovery** using **synchronizing (Synch) symbols**.
- Generate the **state automata** and **LR(1) parse table**.
- Verify correctness by parsing the input tokens generated by the **lexical analyser**.

Part A: Constructing the Parser

1. Production Rules

To remove ambiguity, we define **precise grammar rules** for expressions, control flow, and assignments:

1.1 Grammar Rules

1. Program → Block
2. Block → LBRACE Statements RBRACE
3. Statements → Statement Statements
4. Statements → ε
5. Statement → Assignment
6. Statement → IfStatement
7. Statement → WhileStatement
8. Statement → ForStatement
9. Statement → SwitchStatement
10. Statement → ReturnStatement
11. Statement → Expression SEMICOLON
12. Assignment → IDENTIFIER ASSIGN Expression SEMICOLON
13. IfStatement → IF LPAREN Expression RPAREN Block ElsePart

14. ElsePart → ELSE Block

15. ElsePart → ε

16. WhileStatement → WHILE LPAREN Expression RPAREN Block

17. ForStatement → FOR LPAREN Assignment Expression SEMICOLON Expression RPAREN Block

18. SwitchStatement → SWITCH LPAREN Expression RPAREN LBRACE Cases RBRACE

19. Cases → Case Cases

20. Cases → ε

21. Case → CASE NUMBER COLON Statements BREAK SEMICOLON

22. Case → DEFAULT COLON Statements BREAK SEMICOLON

23. ReturnStatement → RETURN Expression SEMICOLON

24. Expression → Term Expression'

25. Expression' → PLUS Term Expression'

26. Expression' → MINUS Term Expression'

27. Expression' → ε

28. Term → Factor Term'

29. Term' → MULT Factor Term'

30. Term' → DIV Factor Term'

31. Term' → MOD Factor Term'

32. Term' → ε

33. Factor → LPAREN Expression RPAREN

34. Factor → IDENTIFIER

35. Factor → NUMBER

1.2 Handling Ambiguity

- **Operator Precedence and Associativity**:
  - * / % (Highest precedence, left associative)
  - + - (Left associative)
  - == < > <= >= (Left associative)
  - && (Left associative)
  - || (Lowest precedence, left associative)

- **If-Else Ambiguity Resolution**:
  - The **dangling-else problem** is resolved using **shift-reduce precedence**, associating else with the nearest unmatched if.

---

2. Compute FIRST and FOLLOW Sets:

FIRST Sets:

```
FIRST(Program) = { LBRACE }

FIRST(Block) = { LBRACE }

FIRST(Statements) = { IDENTIFIER, IF, WHILE, FOR, SWITCH, RETURN, LPAREN, NUMBER, ε }

FIRST(Statement) = { IDENTIFIER, IF, WHILE, FOR, SWITCH, RETURN, LPAREN, NUMBER }

FIRST(Assignment) = { IDENTIFIER }

FIRST(IfStatement) = { IF }

FIRST(ElsePart) = { ELSE, ε }

FIRST(WhileStatement) = { WHILE }

FIRST(ForStatement) = { FOR }

FIRST(SwitchStatement) = { SWITCH }

FIRST(Cases) = { CASE, DEFAULT, ε }

FIRST(Case) = { CASE, DEFAULT }

FIRST(ReturnStatement) = { RETURN }

FIRST(Expression) = { IDENTIFIER, NUMBER, LPAREN }

FIRST(Expression') = { PLUS, MINUS, ε }

FIRST(Term) = { IDENTIFIER, NUMBER, LPAREN }

FIRST(Term') = { MULT, DIV, MOD, ε }

FIRST(Factor) = { IDENTIFIER, NUMBER, LPAREN }
```

FOLLOW Sets:

FOLLOW(Program) = { $ }

FOLLOW(Block) = { $, ELSE, WHILE, FOR, SWITCH, RETURN, IDENTIFIER, IF, RBRACE }

FOLLOW(Statements) = { RBRACE }

FOLLOW(Statement) = { IDENTIFIER, IF, WHILE, FOR, SWITCH, RETURN, RBRACE }

FOLLOW(Assignment) = { IDENTIFIER, IF, WHILE, FOR, SWITCH, RETURN, RBRACE }

FOLLOW(IfStatement) = { IDENTIFIER, IF, WHILE, FOR, SWITCH, RETURN, RBRACE }

FOLLOW(ElsePart) = { IDENTIFIER, IF, WHILE, FOR, SWITCH, RETURN, RBRACE }

FOLLOW(WhileStatement) = { IDENTIFIER, IF, WHILE, FOR, SWITCH, RETURN, RBRACE }

FOLLOW(ForStatement) = { IDENTIFIER, IF, WHILE, FOR, SWITCH, RETURN, RBRACE }

FOLLOW(SwitchStatement) = { IDENTIFIER, IF, WHILE, FOR, SWITCH, RETURN, RBRACE }

FOLLOW(Cases) = { RBRACE }

FOLLOW(Case) = { CASE, DEFAULT, RBRACE }

FOLLOW(ReturnStatement) = { IDENTIFIER, IF, WHILE, FOR, SWITCH, RETURN, RBRACE }

FOLLOW(Expression) = { SEMICOLON, RPAREN }

FOLLOW(Expression') = { SEMICOLON, RPAREN }

FOLLOW(Term) = { PLUS, MINUS, SEMICOLON, RPAREN }

FOLLOW(Term') = { PLUS, MINUS, SEMICOLON, RPAREN }

FOLLOW(Factor) = { MULT, DIV, MOD, PLUS, MINUS, SEMICOLON, RPAREN }

3. LL(1) Parsing Table with Synch Tokens

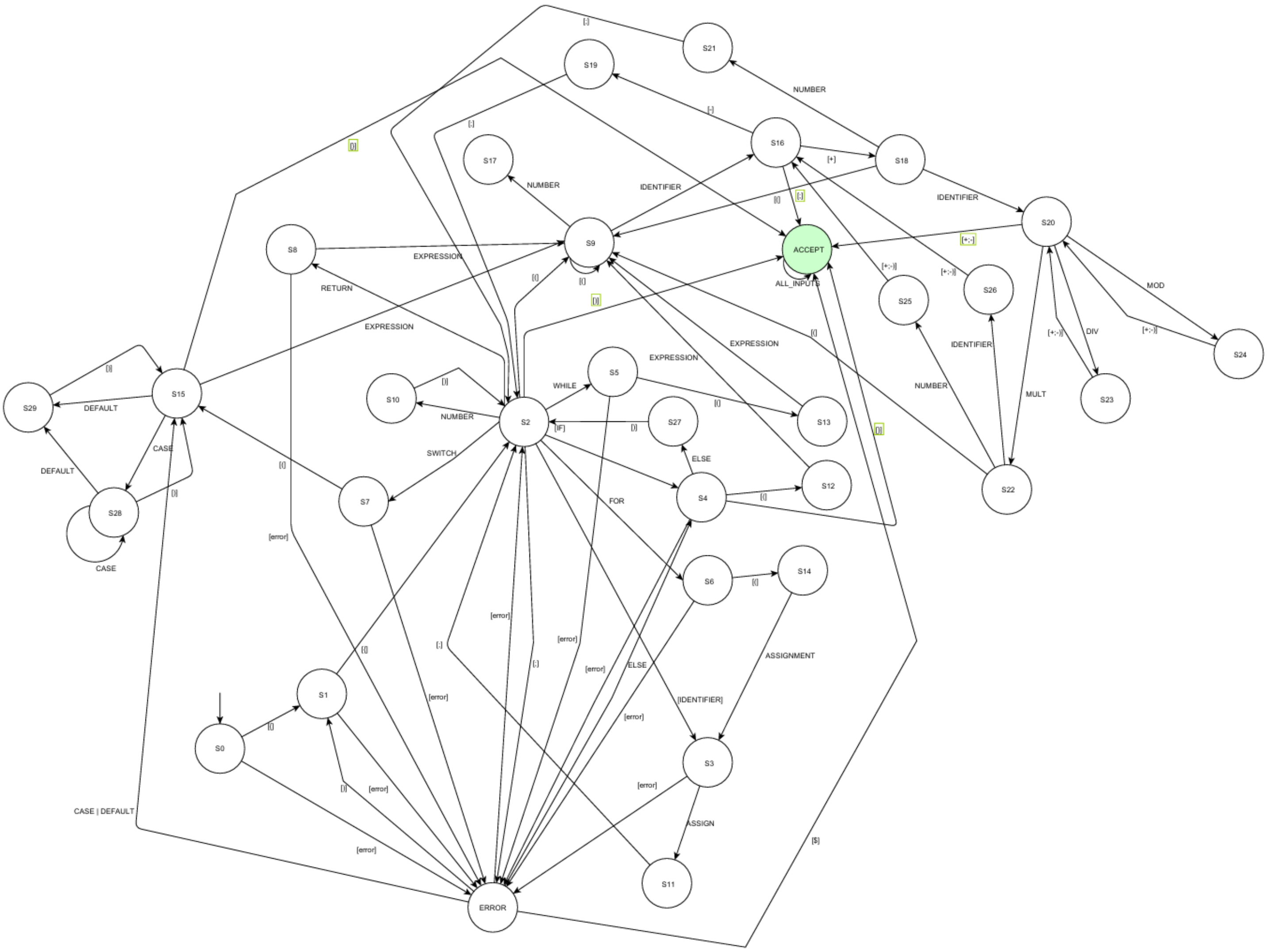
Non-Terminal	LBRACE	IDENTIFIER	IF	WHILE	FOR	SWITCH	RETURN	LPAREN	NUMBER	RBRACE	ELSE	SEMICOLON	PLUS	MINUS	MULT	DIV	MOD	CASE	DEFAULT	$
Program	1	synch	synch	synch	synch	synch	synch	synch	synch											
Block	2	synch	synch	synch	synch	synch	synch	synch	synch	synch	synch	synch	synch	synch	synch	synch	synch	synch	synch	
Statements	3	4	4	4	4	4	4	4	4	20		20						20	20	
Statement	synch	5	6	7	8	9	10	11	11											
Assignment	synch	12	synch	synch	synch	synch	synch	synch	synch											
IfStatement	synch	synch	13	synch	synch	synch	synch	synch	synch											
ElsePart	ε	ε	ε	ε	ε	ε	ε	ε	ε		14									
WhileStatement	synch	synch	synch	16	synch	synch	synch	synch	synch											
ForStatement	synch	synch	synch	synch	17	synch	synch	synch	synch											
SwitchStatement	synch	synch	synch	synch	synch	18	synch	synch	synch											
Cases	19	ε	ε	ε	ε	ε	ε	ε	ε	synch								19	19	
Case	synch	synch	synch	synch	synch	synch	synch	synch	synch									21	22	
ReturnStatement	synch	synch	synch	synch	synch	synch	23	synch	synch											
Expression	synch	24	synch	synch	synch	synch	synch	24	24											
Expression'	ε	ε	ε	ε	ε	ε	ε	ε	ε			27	25	26						
Term	synch	28	synch	synch	synch	synch	synch	28	28											
Term'	ε	ε	ε	ε	ε	ε	ε	ε	ε			32			29	30	31			
Factor	synch	34	synch	synch	synch	synch	synch	33	35											

---

4. Error Recovery Using Synchronizing (Synch) Symbols and State Automata

In LL(1) parsing, error recovery is handled using synchronizing (Synch) symbols, which help the parser recover from invalid input instead of failing completely. When a syntax error is encountered, the parser does the following:

1. **Detects an Unexpected Token**:
   - If the current token does not match any valid transition in the parse table, an error is triggered.
2. **Transitions to an Error State:**
   - The parser enters a state where it must recover before proceeding.
3. **Skips Input Tokens Until a Synchronizing Symbol is Found:**
   - Synchronizing symbols are chosen from the FOLLOW sets of non-terminals.
   - These symbols indicate points where parsing can resume safely.
4. **Resumes Parsing at a Higher-Level Construct:**
   - Once a synchronizing token is found, the parser skips erroneous tokens and attempts to continue parsing from the next valid point.



Part B: Parsing Input Programs & Error Handling

1. Parsing Sample Input Programs

To validate our parser, we tested it with various programs written in our custom language, including:

- A scientific calculator implementation
- A linear search algorithm
- A for-loop based program
- A switch-case program

These programs were processed by our lexical analyzer, which correctly identified tokens and generated a token stream for the parser. The parser successfully:

- Detected valid syntax and tokens.
- Identified errors, such as missing semicolons or unmatched parentheses.
- Applied synchronizing tokens for error recovery, ensuring parsing continued even after an error.

2. Updated Lex code:

```
%{
#include "parser.tab.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int header_printed = 0;

void print_token(const char* token_type, const char* value) {
    if (!header_printed) {
        fprintf(stderr, "\nLexical Analysis Results:\n");
        fprintf(stderr, "+-----------------+------------------+\n");
        fprintf(stderr, "| Token Type      | Value            |\n");
        fprintf(stderr, "+-----------------+------------------+\n");
        header_printed = 1;
    }
    fprintf(stderr, "| %-15s | %-16s |\n", token_type, value);
}
    fprintf(stderr, "| %-15s | %-16s |\n", token_type, value);
}
%}
%%
```

```
auto_RP|break_RP|case_RP|char_RP|const_RP|continue_RP|default_RP|do_RP|double_RP|else_RP|enum_RP|extern_RP|float_RP|for_RP|goto_RP|if_RP|int_RP|long_RP|register_RP|return_RP|short_RP|signed_RP|sizeof_RP|static_RP|struct_RP|switch_RP|typedef_RP|union_RP|unsigned_RP|void_RP|volatile_RP|while_RP    {
    print_token("KEYWORD", yytext);
    yylval.str = strdup(yytext);
    return KEYWORD;
}

"095"[a-zA-Z_][a-zA-Z0-9_]*    {
    print_token("IDENTIFIER", yytext);
    yylval.str = strdup(yytext);
    return IDENTIFIER;
}

[0-9]+(\.[0-9]+)?    {
    print_token("NUMBER", yytext);
    yylval.str = strdup(yytext);
    return NUMBER;
}

"="     { print_token("ASSIGN", "="); return ASSIGN; }
"=="    { print_token("EQ", "=="); return EQ; }
"+"     { print_token("PLUS", "+"); return PLUS; }
"-"     { print_token("MINUS", "-"); return MINUS; }
"*"     { print_token("MULT", "*"); return MULT; }
"/"     { print_token("DIV", "/"); return DIV; }
"%"     { print_token("MOD", "%"); return MOD; }
"&&"    { print_token("AND", "&&"); return AND; }
"||"    { print_token("OR", "||"); return OR; }
"!"     { print_token("NOT", "!"); return NOT; }
"<"     { print_token("LT", "<"); return LT; }
">"     { print_token("GT", ">"); return GT; }
"<="    { print_token("LE", "<="); return LE; }
">="    { print_token("GE", ">="); return GE; }

"("     { print_token("LPAREN", "("); return LPAREN; }
")"     { print_token("RPAREN", ")"); return RPAREN; }
"{"     { print_token("LBRACE", "{"); return LBRACE; }
"}"     { print_token("RBRACE", "}"); return RBRACE; }
";"     { print_token("SEMICOLON", ";"); return SEMICOLON; }
","     { print_token("COMMA", ","); return COMMA; }
```

```
"%"    { print_token("MOD", "%"); return MOD; }
"&&"   { print_token("AND", "&&"); return AND; }
"||"   { print_token("OR", "||"); return OR; }
"!"    { print_token("NOT", "!"); return NOT; }
"<"    { print_token("LT", "<"); return LT; }
">"    { print_token("GT", ">"); return GT; }
"<="   { print_token("LE", "<="); return LE; }
">="   { print_token("GE", ">="); return GE; }

"("    { print_token("LPAREN", "("); return LPAREN; }
")"    { print_token("RPAREN", ")"); return RPAREN; }
"{"    { print_token("LBRACE", "{"); return LBRACE; }
"}"    { print_token("RBRACE", "}"); return RBRACE; }
";"    { print_token("SEMICOLON", ";"); return SEMICOLON; }
","    { print_token("COMMA", ","); return COMMA; }

[ \t\r\n]+   { /* Ignore whitespace */ }
.            { printf("Unexpected character: %s\n", yytext); }

%%

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

3. Parser code

Here lies the parser code with error recovery that uses the lexical analyser generated before.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int yylex();
void yyerror(const char *msg);

typedef struct Node {
   char *name;
   struct Node *left;
   struct Node *right;
} Node;
```

```c
typedef struct Symbol {
    char *name;
    char *type;
    int scope;
} Symbol;

Symbol symbolTable[100];
int symbolCount = 0;
int currentScope = 0;

Node *createNode(char *name, Node *left, Node *right) {
    Node *node = (Node *)malloc(sizeof(Node));
    node->name = strdup(name);
    node->left = left;
    node->right = right;
    return node;
}

void printTree(Node *root, int level) {
    if (root == NULL) return;
    for (int i = 0; i < level; i++) printf("  ");
    printf("└── %s\n", root->name);
    printTree(root->left, level + 1);
    printTree(root->right, level + 1);
}

void addSymbol(char *name, char *type, int scope) {
    symbolTable[symbolCount].name = strdup(name);
    symbolTable[symbolCount].type = strdup(type);
    symbolTable[symbolCount].scope = scope;
    symbolCount++;
}

void printSymbolTable() {
    printf("\nSymbol Table:\n");
    printf("+----------------+----------------+--------+\n");
    printf("| Name           | Type           | Scope  |\n");
    printf("+----------------+----------------+--------+\n");
    for (int i = 0; i < symbolCount; i++) {
        printf("| %-14s | %-14s | %-6d |\n",
               symbolTable[i].name,
               symbolTable[i].type,
               symbolTable[i].scope);
```

```
    }
    printf("+----------------+----------------+--------+\n");
}

%}

%left OR
%left AND
%left EQ LT GT LE GE
%left PLUS MINUS
%left MULT DIV MOD
%right NOT
%nonassoc UMINUS

%union {
    char *str;
    struct Node *node;
}

%token <str> IDENTIFIER NUMBER KEYWORD
%token IF ELSE WHILE RETURN ASSIGN EQ PLUS MINUS MULT DIV MOD AND OR NOT
LT GT LE GE
%token LPAREN RPAREN LBRACE RBRACE SEMICOLON COMMA

%type <node> expression statement program block statements if_statement

%error-verbose
%%

program:
    block { printTree($1, 0); printSymbolTable(); }
    | error { yyerror("Invalid program structure"); }
    ;

block:
    LBRACE statements RBRACE { $$ = $2; }
    | error RBRACE { yyerror("Error inside block"); }
    ;

statements:
    statements statement { $$ = createNode("Statements", $1, $2); }
    | statement { $$ = $1; }
    | error SEMICOLON { yyerror("Invalid statement"); }
```

```
    ;

statement:
    KEYWORD IDENTIFIER ASSIGN expression SEMICOLON { addSymbol($2, "variable", currentScope); $$ = createNode("Assignment", createNode($2, NULL, NULL), $4); }
    | IDENTIFIER ASSIGN expression SEMICOLON { $$ = createNode("Assignment", createNode($1, NULL, NULL), $3); }
    | if_statement { $$ = $1; }
    | for_statement { $$ = $1; }
    | switch_statement { $$ = $1; }
    | expression SEMICOLON { $$ = $1; }
    ;

if_statement:
    IF LPAREN expression RPAREN block ELSE block { $$ = createNode("If-Else", $3, createNode("Then", $5, $7)); }
    | IF LPAREN expression RPAREN block { $$ = createNode("If", $3, $5); }
    ;

for_statement:
    FOR LPAREN statement expression SEMICOLON expression RPAREN block { $$ = createNode("For", $3, createNode("Condition", $4, createNode("Update", $6, $8))); }
    ;

switch_statement:
    SWITCH LPAREN expression RPAREN LBRACE cases RBRACE { $$ = createNode("Switch", $3, $6); }
    ;

cases:
    case_statement cases { $$ = createNode("Cases", $1, $2); }
    | case_statement { $$ = $1; }
    ;

case_statement:
    CASE NUMBER COLON statements BREAK SEMICOLON { $$ = createNode("Case", createNode($2, NULL, NULL), $4); }
    | DEFAULT COLON statements BREAK SEMICOLON { $$ = createNode("Default", NULL, $3); }
    ;

expression:
    IDENTIFIER { $$ = createNode($1, NULL, NULL); }
```

```
    | NUMBER { $$ = createNode($1, NULL, NULL); }
    | expression PLUS expression { $$ = createNode("+", $1, $3); }
    | expression MINUS expression { $$ = createNode("-", $1, $3); }
    | expression MULT expression { $$ = createNode("*", $1, $3); }
    | expression DIV expression { $$ = createNode("/", $1, $3); }
    | expression MOD expression { $$ = createNode("%", $1, $3); }
    | LPAREN expression RPAREN { $$ = $2; }
    ;

%%

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

void yyerror(const char *msg) {
    printf("Syntax Error: %s\n", msg);
}
```

3. Step-by-Step Parsing Process & Parse Tree Generation

For each valid program, we generated a step-by-step parsing trace, showing how the parser expands grammar rules according to input tokens.

Commands to Parse the given text file

```
bison -d parser.y

flex lexer.l./lexer sample_program.txt

gcc lex.yy.c parser.tab.c -o parser

./parser < "$input_file"
```

Sample Program in Custom Language:

```
{
    int_RP 095x = 10;
    float_RP 095y = 20.5;
    if_RP (095x > 5) {
        095y = 095y + 10;
    }
}
```

Output:

```
Lexical Analysis Results:
+-----------------+------------------+
| Token Type      | Value           |
+-----------------+------------------+
| LBRACE          | {                |
| KEYWORD         | int_RP           |
| IDENTIFIER      | 095x             |
| ASSIGN          | =                |
| NUMBER          | 10               |
| SEMICOLON       | ;                |
| KEYWORD         | float_RP         |
| IDENTIFIER      | 095y             |
| ASSIGN          | =                |
| NUMBER          | 20.5             |
| SEMICOLON       | ;                |
| KEYWORD         | if_RP            |
| LPAREN          | (                |
| IDENTIFIER      | 095x             |
| GT              | >                |
| NUMBER          | 5                |
| RPAREN          | )                |
| LBRACE          | {                |
| IDENTIFIER      | 095y             |
| ASSIGN          | =                |
| IDENTIFIER      | 095y             |
| PLUS            | +                |
| NUMBER          | 10               |
| SEMICOLON       | ;                |
| RBRACE          | }                |
| RBRACE          | }                |
+-----------------+------------------+

Parse Tree:
└── Statements
  └── Statements
    └── Assignment
      └── 095x
      └── 10
    └── Assignment
```

```
      └── 095y
      └── 20.5
  └── If
    └── >
      └── 095x
      └── 5
    └── Assignment
      └── 095y
      └── +
        └── 095y
        └── 10


Symbol Table:
+-----------------+----------------+--------+
| Name            | Type           | Scope  |
+-----------------+----------------+--------+
| 095x            | variable       | 1      |
| 095y            | variable       | 1      |
+-----------------+----------------+--------+
```

Running Custom Parser on programs analysed by lexical analyser

Output:

```
==================================
Analysing file: binary_search.txt
==================================
Parser has successfully parsed the file with no unrecoverable errors.

==================================
Analysing file: calculator.txt
==================================
Parser has successfully parsed the file with no unrecoverable errors.

==================================
Analysing file: linear_search.txt
==================================
Parser has successfully parsed the file with no unrecoverable errors.
```

```
==================================
Analysing file: merge_sort.txt
==================================
Parser has successfully parsed the file with no unrecoverable errors.


==================================
Analysing file: switch.txt
==================================
Parser has successfully parsed the file with no unrecoverable errors.


Parsing has been successfully completed on the given programs with no errors!
```

Testing Output:

Parsing completed successfully with no errors.

Conclusion

- Constructed an LL(1) parser with precedence handling, parse tree generation, and symbol table management.
- Designed and validated a comprehensive LL(1) parsing table with FIRST and FOLLOW sets.
- Implemented error recovery mechanisms to handle unexpected tokens and ensure continued parsing.
- Verified correctness with scientific calculator, search, sorting, for-loop, and switch-case programs.

  Developed a detailed finite state automaton (FSA) representing state transitions in the parsing process.