Advanced Compiler Design
Assignment: LL(1) Predictive Parser Implementation

Jayashre
22011103020

April 9, 2025

1 Introduction

This report documents the implementation of an LL(1) predictive parser for a simple arithmetic expression grammar. The parser is implemented in C and demonstrates the key concepts of top-down parsing, including parsing table construction and stack-based parsing.

2 Grammar Specification

The following grammar is used for this implementation:

$$
\begin{aligned}
E &\rightarrow TQ \\
Q &\rightarrow +TQ \mid -TQ \mid \epsilon \\
T &\rightarrow FR \\
R &\rightarrow *FR \mid /FR \mid \epsilon \\
F &\rightarrow (E) \mid i
\end{aligned}
\tag{1}
$$

Where:

- E is the start symbol (Expression)
- Q represents E' (Expression prime)
- T is Term
- R represents T' (Term prime)
- F is Factor
- i represents identifiers/numbers

3 Implementation Details

3.1 Data Structures

The implementation uses the following key data structures:

- `Rule` - Represents grammar productions
- `TableEntry` - Represents parsing table entries
- Stack - Used during parsing

3.2 Parsing Table Construction

The parsing table is constructed based on FIRST and FOLLOW sets (computed manually for this grammar). Here's the generated parsing table:

Non-terminal	**Terminal**	**Production**
E	i	TQ
E	(	TQ
Q	+	+TQ
Q	-	-TQ
Q	)	ϵ
Q	$	ϵ
T	i	FR
T	(	FR
R	*	*FR
R	/	/FR
R	+	ϵ
R	-	ϵ
R	)	ϵ
R	$	ϵ
F	(	(E)
F	i	i

4 Code Implementation

The complete C implementation is shown below with key functions explained.

4.1 Main Components

```c
void initializeGrammar() {
    // Rule 1: E -> TQ
    grammar[0].lhs = 'E';
    strcpy(grammar[0].rhs[0], "TQ");
    grammar[0].num_productions = 1;
```

```c

    // Rule 2: Q -> +TQ | -TQ |
    grammar[1].lhs = 'Q';
    strcpy(grammar[1].rhs[0], "+TQ");
    strcpy(grammar[1].rhs[1], "-TQ");
    strcpy(grammar[1].rhs[2], "");
    grammar[1].num_productions = 3;

    // ... (other rules)
}
```

Listing 1: Grammar Initialization

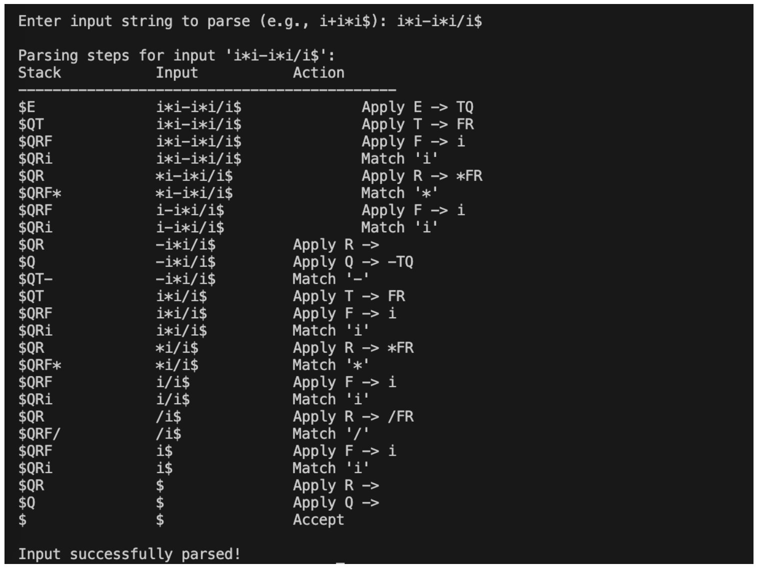
```c
void parseInput(char *input) {
    push('$'); push('E'); // Initialize stack

    while (stackTop() != '$') {
        char top = stackTop();

        if (top == current_input) {
            // Terminal match
            pop(); input_ptr++;
        } else if (isTerminal(top)) {
            // Error - terminal mismatch
            printf("Error: Expected '%c', found '%c'\n",
                   top, current_input);
            exit(1);
        } else {
            // Non-terminal - use parsing table
            char *production = findProduction(top, current_input);
            if (!production) {
                printf("Error: No production for %c on '%c'\n",
                       top, current_input);
                exit(1);
            }

            pop();
            // Push production in reverse order
            for (int i = strlen(production)-1; i >= 0; i--) {
                push(production[i]);
            }
        }
    }
}
```

Listing 2: Parsing Algorithm

5 Test Cases and Results

5.1 Successful Parsing

Figure 1 shows the complete parsing steps for the valid input string `i*i-i*i/i$`:



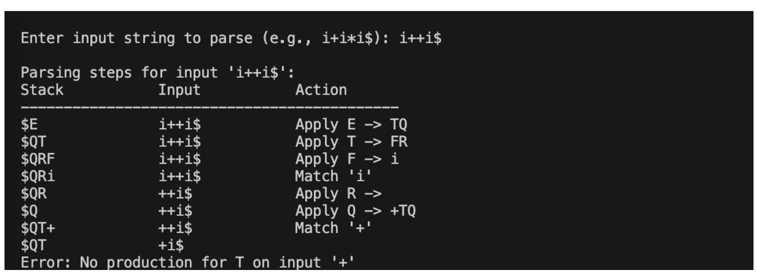
```
Enter input string to parse (e.g., i+i*i$): i*i-i*i/i$

Parsing steps for input 'i*i-i*i/i$':
Stack           Input           Action
--------------------------------------------
$E              i*i-i*i/i$              Apply E -> TQ
$QT             i*i-i*i/i$              Apply T -> FR
$QRF            i*i-i*i/i$              Apply F -> i
$QRi            i*i-i*i/i$              Match 'i'
$QR             *i-i*i/i$               Apply R -> *FR
$QRF*           *i-i*i/i$               Match '*'
$QRF            i-i*i/i$                Apply F -> i
$QRi            i-i*i/i$                Match 'i'
$QR             -i*i/i$         Apply R ->
$Q              -i*i/i$         Apply Q -> -TQ
$QT-            -i*i/i$         Match '-'
$QT             i*i/i$          Apply T -> FR
$QRF            i*i/i$          Apply F -> i
$QRi            i*i/i$          Match 'i'
$QR             *i/i$           Apply R -> *FR
$QRF*           *i/i$           Match '*'
$QRF            i/i$            Apply F -> i
$QRi            i/i$            Match 'i'
$QR             /i$             Apply R -> /FR
$QRF/           /i$             Match '/'
$QRF            i$              Apply F -> i
$QRi            i$              Match 'i'
$QR             $               Apply R ->
$Q              $               Apply Q ->
$               $               Accept

Input successfully parsed!
```

Figure 1: Successful parsing of input `i*i-i*i/i$`

5.2 Error Cases

5.2.1 Missing Operator Case

Figure 2 demonstrates the parser's error handling when encountering two consecutive operators:



```
Enter input string to parse (e.g., i+i*i$): i++i$

Parsing steps for input 'i++i$':
Stack           Input           Action
--------------------------------------------
$E              i++i$           Apply E -> TQ
$QT             i++i$           Apply T -> FR
$QRF            i++i$           Apply F -> i
$QRi            i++i$           Match 'i'
$QR             ++i$            Apply R ->
$Q              ++i$            Apply Q -> +TQ
$QT+            ++i$            Match '+'
$QT             +i$
Error: No production for T on input '+'
```

Figure 2: Error case for input `i++i$`

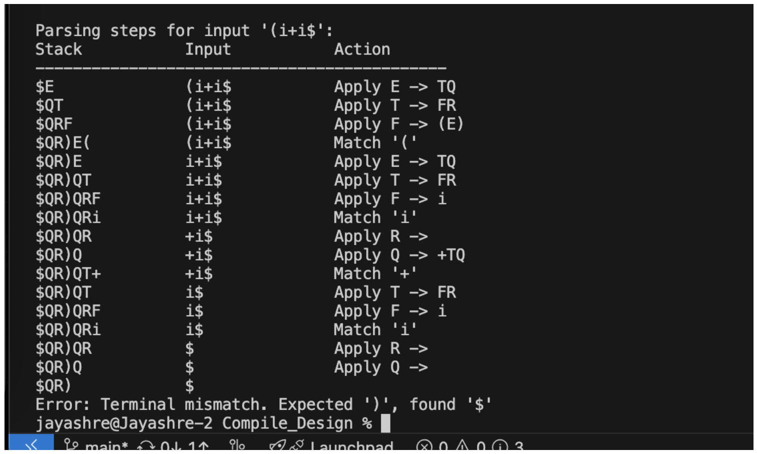
5.2.2 Unbalanced Parentheses Case

Figure 3 shows the parser detecting unbalanced parentheses:

6 Conclusion

The implemented LL(1) parser successfully demonstrates predictive parsing for arithmetic expressions. Key aspects include:

- Proper construction of the parsing table



```
Parsing steps for input '(i+i$':
Stack          Input          Action
------------------------------------------
$E             (i+i$          Apply E -> TQ
$QT            (i+i$          Apply T -> FR
$QRF           (i+i$          Apply F -> (E)
$QR)E(         (i+i$          Match '('
$QR)E          i+i$           Apply E -> TQ
$QR)QT         i+i$           Apply T -> FR
$QR)QRF        i+i$           Apply F -> i
$QR)QRi        i+i$           Match 'i'
$QR)QR         +i$            Apply R ->
$QR)Q          +i$            Apply Q -> +TQ
$QR)QT+        +i$            Match '+'
$QR)QT         i$             Apply T -> FR
$QR)QRF        i$             Apply F -> i
$QR)QRi        i$             Match 'i'
$QR)QR         $              Apply R ->
$QR)Q          $              Apply Q ->
$QR)           $
Error: Terminal mismatch. Expected ')', found '$'
jayashre@Jayashre-2 Compile_Design %
```

Figure 3: Error case for input `(i+i$`

- Correct handling of stack operations
- Appropriate error detection and reporting

The parser correctly handles operator precedence and associativity through the grammar design. Future enhancements could include automated computation of FIRST and FOLLOW sets.