Name: Zelalem Getahun **ID: 1508462**

Bahir Dar University

Principles of Compiler Design on Syntax Analysis

Individual Assignment-1

Table of Contents

1. (Theory) Predictive Parsing Explained with Example

Predictive parsing is a **top-down parsing technique** used in compilers to perform **syntax analysis** of source programs. It attempts to construct the parse tree starting from the start symbol and works its way toward the input string. The key idea is that the parser "predicts" which production rule to use based on the **current non-terminal** on the stack and the **next input token**.

This type of parser is widely known as an **LL(1) parser**, where:

- **L** → Input is scanned from **Left to right**
- **L** → Produces a **Leftmost derivation**
- **1** → Uses **one lookahead symbol** to make decisions

So, an LL(1) parser reads the source program from left to right, expands the leftmost non-terminal first, and uses only one lookahead token to select the appropriate grammar rule.

Why Predictive Parsing Is Powerful

Traditional top-down parsers sometimes require **backtracking**, meaning they try one production, fail, and then go back to try another. This is slow and inefficient.

Predictive parsing **eliminates backtracking completely**.
Instead, it uses **FIRST** and **FOLLOW** sets to construct a **parsing table**. This table directly tells the parser which production to choose in any given situation, making parsing:

- faster
- deterministic
- efficient
- easier to implement

Key Characteristics of Predictive Parsing

- ✓ **Top-down** parsing approach
- ✓ **Table-driven** mechanism using a predictive parsing table
- ✓ Uses **one lookahead token**
- ✓ **No backtracking required**
- ✓ **Efficient and deterministic**
- ✓ Well-suited for programming language compilers

Example Grammar

Consider the following grammar for simple arithmetic expressions:

```
E  → T E'
E' → + T E' | ε
T  → F T'
T' → * F T' | ε
F  → (E) | id
```

This grammar represents expressions that include:

- identifiers (id)
- addition (+)
- multiplication (*)
- parentheses

Example Input

```
id + id
```

How Predictive Parsing Works

Step 1 - Start Parsing

The parser begins with the start symbol:

```
Stack: E
Input: id + id $
```

Step 2 - Look Ahead

The lookahead token is **id**.

Step 3 - Select Production

Based on the parsing table (constructed using FIRST and FOLLOW sets):

- Since the top of stack is `E`
- And lookahead is `id`
- The rule chosen is:

E → T E'

Step 4 - Expand Further

To derive `T`, the parser again looks at the input (`id`) and selects:

T → F T'

Then for `F`, since lookahead is still `id`, we use:

F → id

So far the derivation is:

E ⇒ T E'
T ⇒ F T'
F ⇒ id

Handling + Operator

After matching `id`, the next input token becomes `+`.

At this point the parser uses:

E' → + T E'

Then again:

T → F T'
F → id

When no more operators remain, the parser uses ε-productions to finish:

E' → ε
T' → ε

Result

The string `id + id` is successfully parsed **without any backtracking**, proving the power of predictive parsing.

Advantages of Predictive Parsing

✓ Deterministic and fast
✓ Eliminates backtracking → improves performance
✓ Very suitable for compiler front-ends
✓ Simple to implement using:

- stack-based parser
- table-driven parser

Limitations of Predictive Parsing

✗ Grammar **must be LL(1)**
✗ Cannot handle **left recursion** (must be removed first)
✗ Cannot handle certain ambiguous or complex grammars
✗ Sometimes grammar needs to be transformed using:

- left recursion elimination
- left factoring

Conclusion

Predictive parsing is one of the most important parsing methods used in modern compilers. By using **lookahead tokens**, along with **FIRST and FOLLOW sets**, it constructs a parsing table that guides every decision during syntax analysis. This makes the process **deterministic, efficient, and highly reliable**, which is why predictive (LL(1)) parsers are widely used in real compiler design, especially in the syntax analysis phase of programming languages.

2. (C++) Write a Function to Count Identifiers in a Source Code String

Answer

Identifiers are names given to variables, functions, etc. A valid identifier:

- Starts with a letter or underscore
- Followed by letters, digits, or underscore

C++ Program

```
#include <iostream>

#include <cctype>

using namespace std;


bool isIdentifierStart(char c) {

   return isalpha(c) || c == '_';

}


bool isIdentifierChar(char c) {

   return isalnum(c) || c == '_';

}


int countIdentifiers(string code) {

   int count = 0;

   int n = code.length();


   for(int i = 0; i < n; i++) {

      if(isIdentifierStart(code[i])) {

         int j = i + 1;
```

```
            while(j < n && isIdentifierChar(code[j])) {
               j++;
            }

            count++;
            i = j - 1;
         }
      }

      return count;
   }

   int main() {
      string code = "int sum = value1 + num2; float _avg = sum2;";
      cout << "Identifiers: " << countIdentifiers(code);
   }
```

This program scans the source code string and counts how many valid identifiers appear.

3. (Problem Solving) Construct a Parse Tree

Given Grammar

S → 1S0 | 10

Input String

1100

Leftmost Derivation

S ⇒ 1 S 0

⇒ 1 (1 S 0) 0

⇒ 1 (1 0) 0

So,

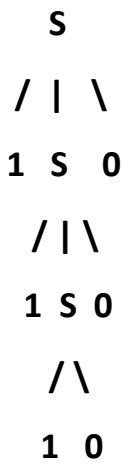
S → 1S0

S → 1(1S0)0

S → 1(10)0

String generated = 1100

Parse Tree



Thus, the grammar successfully generates the string **1100**.