Bahir Dar University BiT

Faculty of Computing Department

Of Software Engineering

Course: Principle of Compiler Design

By: Zefasil Mulu (1508437)

Submitted to: Lecturer Wondimu

Submission date: 27/04/2018 E.C Bahir Dar, Ethiopia

1. Theory: Explain the concept of predictive parsing.

Predictive parsing is a top-down parsing technique in which the parser **predicts** which production rule to apply by examining the next few tokens in the input stream, without backtracking. It is used for **LL(k)** grammars, where *k* indicates the number of lookahead tokens. Predictive parsers work efficiently for grammars that are **non-left-recursive** and **left-factored**.

Key characteristics:

- **Deterministic:** At each step, only one production rule is possible.
- **Lookahead tokens:** Uses a parsing table constructed from **FIRST** and **FOLLOW** sets.
- **Implementation methods:**
    - **Recursive descent parsing:** Each non-terminal is implemented as a function.
    - **Table-driven LL(1) parsing:** Uses a stack and parsing table.

Example:
Grammar:

text

E → T E’

E’ → + T E’ | ε

T → F T’

T’ → * F T’ | ε

F → (E) | id

This grammar is suitable for predictive parsing because it is LL(1).

2. C++ Implementation: Function to count identifiers in a source code string

```
#include <iostream>

#include <string>

#include <cctype>
```

```
using namespace std;

/**
 * Function: countIdentifiers
 * Purpose: Counts valid C/C++ style identifiers in a source string.
 * Rules:
 *   - Identifier must start with a letter (a-z, A-Z) or underscore (_).
 *   - Subsequent characters can be letters, digits, or underscores.
 */
int countIdentifiers(const string& source) {
    int count = 0;
    bool insideIdentifier = false;

    for (size_t i = 0; i < source.length(); ++i) {
        char ch = source[i];

        // Start of a new identifier
        if (isalpha(ch) || ch == '_') {
            if (!insideIdentifier) {
                ++count;
                insideIdentifier = true;
            }
        }
        // Continuation of existing identifier
        else if (isdigit(ch) && insideIdentifier) {
```

```
      // Continue within the same identifier
    }
    // End of identifier
    else {
      insideIdentifier = false;
    }
  }

  return count;
}

int main() {
  // Test case
  string code = "int main() { int x1 = 10; float _value = 20.5; result = x1 + _value; }";
  int identifierCount = countIdentifiers(code);
  cout << "Source Code: \"" << code << "\"" << endl;
  cout << "Number of identifiers found: " << identifierCount << endl;

  return 0;
}
```

Sample Output:

text

```
Source Code: "int main() { int x1 = 10; float _value = 20.5; result = x1 + _value; }"

Number of identifiers found: 7
```

Identifiers detected: main, x1, _value, result, x1, _value (note: duplicates counted as separate occurrences).

3. Problem-Solving: Parse tree for grammar S → 1S0 | 10 and input "1100"

Grammar:

text

S → 1S0

S → 10

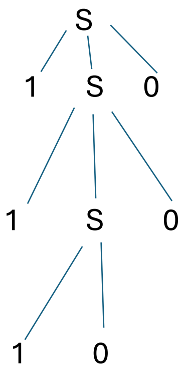
Input string: 1100

Derivation steps:

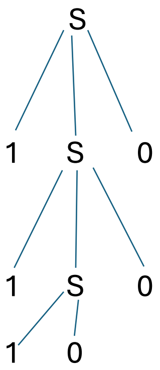
1. S ⇒ 1S0
2. ⇒ 1(1S0)0
3. ⇒ 1(1(10)0)0
4. ⇒ 1100

Parse Tree:

text



Textual representation: text



Explanation:

The parse tree shows how the string 1100 is generated using the given grammar.

- First S expands to 1S0.
- The second S expands to 1S0.
- The third S expands to 10.

This results in the sequence 1 1 10 0 0 = 1100.