



**Bahir Dar University Bahir Dar Institute of Technology  
Faculty of Computing DEPARTMENT OF INFORMATION  
TECHNOLOGY**

**Principles of Compiler Design on Syntax Analysis  
Individual Assignment  
Assignment 01**

**Prepared by:**

**NAME: - YARED MESFIN**

**ID: 1411368**

# **Question 1 Explain the concept of predictive parsing.**

## **Definition**

Predictive parsing is an efficient form of Top Down parsing used in syntax analysis. Its primary defining characteristic is that it does not require backtracking. Instead of guessing which grammar rule to apply and potentially failing a predictive parser determines the correct production rule to apply solely by looking at the current lookahead symbol.

## **How It Works**

The parser attempts to construct a Parse Tree from the root (Start Symbol) down to the leaves. It operates on a specific class of grammars known as **LL(k)** grammars (usually LL(1)).

- **Lookahead:** The parser peekes at the next incoming token.
- **Decision:** Based on the current Non Terminal it is trying to expand and that lookahead token it consults a set of rules to uniquely identify which production to use.

## **Key Characteristics**

- **Deterministic:** Because the parser knows exactly which path to take at every step the time complexity is linear typically  $O(n)$  where  $n$  is the length of the input string. This makes it much faster than general parsing methods.
- **LL(1) Parsing:** Most predictive parsers are classified as LL(1):
  - **L:** Scans input from Left to right.
  - **L:** Produces a Leftmost derivation.
  - **(1):** Uses **1** symbol of lookahead to make parsing decisions.

## Implementation Approaches

There are two main ways to implement a predictive parser:

### 1. Recursive Descent Parser:

- This is a code based approach where every Non Terminal in the grammar corresponds to a function (e.g., void Statement() void Expression()).
- The functions call each other recursively to match the input against the grammar.

### 2. Non-Recursive (Table-Driven) Parser:

- This approach uses an explicit **Stack** data structure and a **Parsing Table**.
- The parsing table is a 2D array constructed using **FIRST** and **FOLLOW** sets. The parser looks up Table to decide whether to push new symbols to the stack or pop matching symbols.

## Prerequisites for the Grammar

For predictive parsing to function correctly the underlying grammar must be unambiguous and satisfy two specific conditions:

- **No Left Recursion:** The grammar cannot have rules like  $A \rightarrow A\alpha$ . If present, this causes an infinite loop in the parser. Left recursion must be eliminated during the grammar analysis phase.
- **Left Factored:** The grammar cannot have rules that start with the same symbol (e.g.,  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ ). This causes ambiguity for the lookahead. The grammar must be "left factored" so the parser can make a distinct choice.

## Question 2: Write a function to count identifiers in a source code string.

Here's a C++ function to count identifiers in a source code string according to typical C/C++ identifier rules:

```
#include <iostream>
#include <string>
#include <cctype>
#include <unordered_set>

int countIdentifiers(const std::string& source) {
    // Set of C++ keywords (partial list, can be expanded)
    static const std::unordered_set<std::string> keywords = {
        "alignas", "alignof", "and", "and_eq", "asm", "auto", "bitand", "bitor",
        "bool", "break", "case", "catch", "char", "char8_t", "char16_t", "char32_t",
        "class", "compl", "concept", "const", "const_cast", "constexpr", "consteval",
        "constinit", "continue", "co_await", "co_return", "co_yield", "decltype",
        "default", "delete", "do", "double", "dynamic_cast", "else", "enum",
        "explicit", "export", "extern", "false", "float", "for", "friend", "goto",
        "if", "inline", "int", "long", "mutable", "namespace", "new", "noexcept",
        "not", "not_eq", "nullptr", "operator", "or", "or_eq", "private", "protected"
    ,
        "public", "register", "reinterpret_cast", "requires", "return", "short",
        "signed", "sizeof", "static", "static_assert", "static_cast", "struct",
        "switch", "template", "this", "thread_local", "throw", "true", "try",
        "typedef", "typeid", "typename", "union", "unsigned", "using", "virtual",
        "void", "volatile", "wchar_t", "while", "xor", "xor_eq"
    };

    int count = 0;
    size_t i = 0;
    size_t n = source.length();

    // Skip whitespace, comments, and string literals
    auto skipWhitespace = [&]() {
        while (i < n && std::isspace(source[i])) {
            i++;
        }
    }
```

```

    }

};

auto skipSingleLineComment = [&]() {
    i += 2; // Skip "://" 
    while (i < n && source[i] != '\n') {
        i++;
    }
};

auto skipMultiLineComment = [&]() {
    i += 2; // Skip "/*"
    while (i + 1 < n && !(source[i] == '*' && source[i + 1] == '/')) {
        i++;
    }
    if (i + 1 < n) {
        i += 2; // Skip "*/"
    }
};

auto skipStringLiteral = [&]() {
    char quote = source[i];
    i++; // Skip opening quote

    while (i < n) {
        if (source[i] == '\\') {
            i += 2; // Skip escape sequence
            continue;
        }
        if (source[i] == quote) {
            i++; // Skip closing quote
            break;
        }
        i++;
    }
};

auto isIdentifierChar = [] (char c, bool firstChar) {
    if (firstChar) {

```

```

        return std::isalpha(static_cast<unsigned char>(c)) || c == '_';
    } else {
        return std::isalnum(static_cast<unsigned char>(c)) || c == '_';
    }
};

while (i < n) {
    skipWhitespace();
    if (i >= n) break;

    // Handle comments
    if (i + 1 < n && source[i] == '/' && source[i + 1] == '/') {
        skipSingleLineComment();
        continue;
    }

    if (i + 1 < n && source[i] == '/' && source[i + 1] == '*') {
        skipMultiLineComment();
        continue;
    }

    // Handle string and character literals
    if (source[i] == '"' || source[i] == '\'') {
        skipStringLiteral();
        continue;
    }

    // Check for identifier start
    if (isIdentifierChar(source[i], true)) {
        size_t start = i;
        std::string identifier;

        // Extract the full identifier
        while (i < n && isIdentifierChar(source[i], false)) {
            identifier += source[i];
            i++;
        }

        // Check if it's not a keyword
    }
}

```

```

        if (keywords.find(identifier) == keywords.end()) {
            count++;
        }

        continue;
    }

    i++; // Skip other characters
}

return count;
}

// Example usage and test
int main() {
    // Test cases
    std::string testCode1 = R"(

        int main() {
            int x = 10;
            int y = x + 5;
            std::cout << "Result: " << y << std::endl;
            return 0;
        }
    )";

    std::string testCode2 = R"(

        // This is a comment
        class MyClass {
    private:
        int memberVar;
    public:
        void myFunction(int param) {
            int localVar = param * 2;
            if (localVar > 10) {
                return;
            }
        }
    };
)" ;
}

```

```

std::string testCode3 = R"(

    /* Multi-line comment
       with identifiers that shouldn't be counted */

    float calculate_area(float radius) {
        const float PI = 3.14159;
        float area = PI * radius * radius;
        return area;
    }
)";

std::cout << "Test 1 - Basic code:" << std::endl;
std::cout << "Identifiers found: " << countIdentifiers(testCode1) << std::endl;
std::cout << std::endl;

std::cout << "Test 2 - Class with members:" << std::endl;
std::cout << "Identifiers found: " << countIdentifiers(testCode2) << std::endl;
std::cout << std::endl;

std::cout << "Test 3 - Function with constants:" << std::endl;
std::cout << "Identifiers found: " << countIdentifiers(testCode3) << std::endl;

return 0;
}

```

## Key Features:

1. **Identifier Rules:** Follows C++ standard starts with letter/underscore contains letters/digits/underscores.
2. **Keyword Filtering:** Excludes C++ keywords from the count.
3. **Comment Handling:** Skips both single-line (//) and multi-line /\* \*/ comments.
4. **String Literal Handling:** Ignores content inside string and character literals.
5. **Escape Sequence Awareness:** Properly handles escape sequences in strings.

## **Limitations and Considerations:**

1. **Preprocessor Directives:** This implementation doesn't handle preprocessor directives. Identifiers in #define, #include etc. would be counted.
2. **Full Keyword List:** The keyword list can be expanded for completeness.
3. **Standard Library Types:** The current implementation counts standard library types (like std::cout) as identifiers. You might want to filter them depending on your needs.
4. **Context Awareness:** Doesn't understand scopes or namespaces just counts all valid identifiers

## **Question 3 Draw the parse tree for "1100"**

### **Grammar Given:**

1.  $S \rightarrow 1S0$
2.  $S \rightarrow 10$

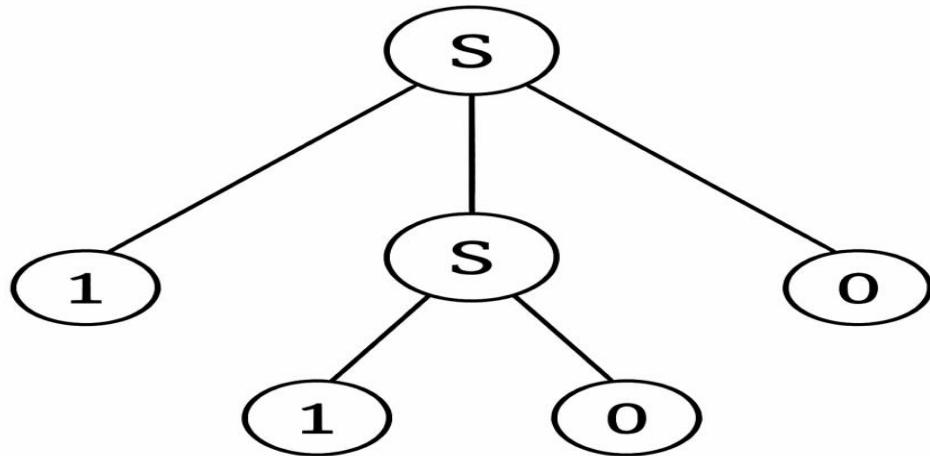
**String to derive:** 1100

### **Step by Step Derivation:**

To get 1100, we start with S:

1. Apply  $S \rightarrow 1S0$  (Now we have 1, a middle S, and 0)
2. Inside that middle S apply  $S \rightarrow 10$
3. Result:  $1(10)0 = 1100$

## The Parse Tree:



### Explanation of the tree:

- The **Root** is the starting symbol SS.
- The first branch uses the recursive rule  $S \rightarrow 1S0$ .
- The inner SS node then uses the "base case" rule  $S \rightarrow 10$  to complete the string.
- Reading the "leaves" (the bottom-most characters) from left to right gives you: **1, 1, 0**.