Bahirdar University Faculty of Computing Software Engineering Copiler Design Assignment 1 2018 E.C

Abebaw Abebe --------BDU1504734

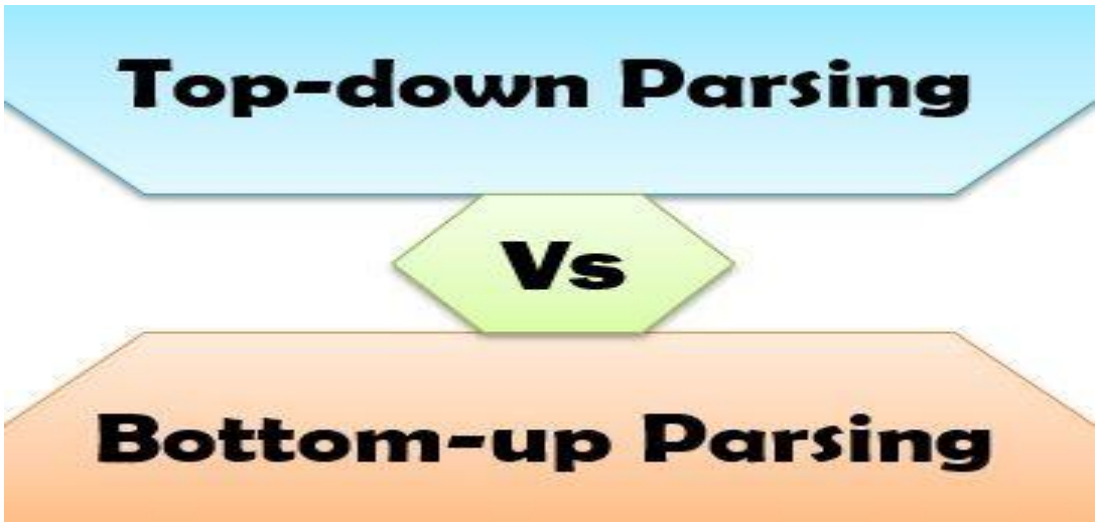
Submitted to: M.r Wondimu B.
Subition Dtae: 27-04-2018

Table of Contents Pages

Difference Between Top-Down and Bottom-Up Parsing

1. Introduction to Syntax Analysis

Syntax analysis, also known as parsing, is the second phase of a compiler. In this phase, the compiler checks whether the sequence of tokens generated by the lexical analyzer follows the grammatical rules of the programming language. The syntax analyzer works using a Context-Free Grammar (CFG), constructs a parse tree or syntax tree to represent the grammatical structure of the input, and reports syntax errors if the structure of the program is invalid. Based on how the parse tree is constructed, parsing techniques are mainly classified into two categories: Top-Down Parsing and Bottom-Up Parsing.



2. Top-Down Parsing

2.1 Definition

Top-down parsing is a parsing technique that begins with the start symbol of the grammar and attempts to derive the given input string by repeatedly expanding non-terminal symbols. In this approach, the parse tree is constructed from the root toward the leaves. Essentially, the parser tries to produce the leftmost derivation of the input string.

2.2 Working Principle

In top-down parsing, the parsing process starts with the start symbol (S). The parser repeatedly replaces non-terminals with the right-hand side of appropriate production rules and compares the derived symbols with the input tokens from left to right. If at any point the derived symbols do not match the input tokens, parsing fails. The construction of the parse tree proceeds from top to bottom, while the input string is scanned from left to right.

2.3 Example

Consider the grammar $S \rightarrow aSb \mid \varepsilon$ and the input string aabb. The top-down parser starts with the start symbol S and applies the production rule $S \rightarrow aSb$. The non-terminal S is expanded again using the same rule, and finally S is replaced by ε. After completing these steps, the derived string becomes aabb, which matches the input string, so parsing succeeds.

2.4 Types of Top-Down Parsers

One common type of top-down parser is Recursive Descent Parsing. It uses a set of recursive procedures, where each non-terminal in the grammar has a corresponding procedure. This method is simple and easy to implement but may involve backtracking. Another type is Predictive Parsing, also known as LL parsing. Predictive parsers use a parsing table and do not require backtracking, but the grammar must satisfy the LL(1) condition.

2.5 Limitations of Top-Down Parsing

Top-down parsing has several limitations. It cannot handle left-recursive grammars such as $A \rightarrow A\alpha$. It also struggles with ambiguous grammars and grammars that require excessive backtracking, which can make parsing inefficient or impossible.

2.6 Advantages of Top-Down Parsing

Despite its limitations, top-down parsing is simple and intuitive. It is easy to implement manually and is especially useful for small programming languages. Another advantage is that the parse tree is constructed directly during the parsing process.

3. Bottom-Up Parsing

3.1 Definition

Bottom-up parsing is a parsing technique that starts with the input string and attempts to reduce it to the start symbol by reversing the production rules of the grammar. In this method, the parse tree is built from the leaves up to the root. It constructs the rightmost derivation of the input string in reverse order.

3.2 Working Principle

In bottom-up parsing, the parser begins with the input tokens and continuously looks for substrings that match the right-hand side of production rules. When such a substring is found, it is replaced with the corresponding non-terminal symbol, a process known as reduction. This process continues until the entire input string is reduced to the start symbol. The parse tree is constructed from bottom to top, while input scanning is performed from left to right.

3.3 Example

Using the grammar $S \rightarrow aSb \mid \varepsilon$ and the input string aabb, the bottom-up parser first recognizes ε, then reduces substrings according to the grammar rules. Through a sequence of reductions, the input string is finally reduced to the start symbol S, indicating successful parsing.

3.4 Types of Bottom-Up Parsers

One important bottom-up parsing method is Shift-Reduce Parsing, which uses a stack. The parser performs two main operations: shift, where input symbols are pushed onto the stack, and reduce, where symbols on the stack are replaced using production rules. Another powerful category of bottom-up parsers is LR parsers, which include LR(0), SLR(1), CLR(1), and LALR(1). These parsers are widely used in practical compiler construction.

3.5 Advantages of Bottom-Up Parsing

Bottom-up parsing can handle left-recursive grammars and supports a larger class of grammars compared to top-down parsing. It is more powerful and flexible and is commonly used in real-world compilers such as YACC.

3.6 Limitations of Bottom-Up Parsing

Although powerful, bottom-up parsing is more complex to implement. The parsing tables required are usually large, and the overall process is harder to understand compared to top-down parsing.

4. Key Differences Between Top-Down and Bottom-Up Parsing

Top-down parsing starts from the start symbol, while bottom-up parsing begins with the input string. In top-down parsing, the parse tree is constructed from the root to the leaves, whereas in bottom-up parsing it is built from the leaves to the root. Top-down parsing uses leftmost derivation, while bottom-up parsing constructs the rightmost derivation in reverse. Top-down parsing cannot handle left recursion, but bottom-up parsing can. In terms of grammar power, top-down parsing is less powerful, while bottom-up parsing is more powerful. Implementation of top-down parsing is relatively simple, whereas bottom-up parsing is more complex. Common top-down parsers include LL and recursive descent parsers, while common bottom-up parsers include LR and shift-reduce parsers. In practice, bottom-up parsing is widely used in large compilers, while top-down parsing is less common.

Feature	Top-Down Parsing	Bottom-Up Parsing
Starting point	Start symbol	Input string
Parse tree direction	Root → Leaves	Leaves → Root
Derivation	Leftmost derivation	Rightmost derivation (reverse)
Handles left recursion	No	Yes
Grammar power	Less powerful	More powerful
Implementation	Simple	Complex
Common parsers	LL, Recursive Descent	LR, Shift-Reduce
Used in practice	Rare in large compilers	Widely used

5. (C++): Program to Check if a String Contains Balanced Curly Braces {}

5.1 Problem Understanding

A string is said to have balanced curly braces if every opening brace { has a corresponding closing brace }, the braces are closed in the correct order, and at no point does a closing brace appear before its matching opening brace. For example, strings like {}, {{}}, { { } { } }, and {} are balanced, while {, }, and }{ are not balanced.

5.2 Logic Behind the Solution

To solve this problem, a stack is used, which follows the Last In First Out (LIFO) principle. Every opening brace { is pushed onto the stack, and every closing brace } must match a previously pushed {. If a closing brace appears when the stack is empty, the string is unbalanced. After processing the entire string, the stack must be empty for the braces to be considered balanced. This approach is commonly used in syntax analysis to check matching symbols.

5.3 C++ Program

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
   string input;
   stack<char> s;

   cout << "Enter a string: ";
   cin >> input;

   bool balanced = true;

   for (char ch : input) {
      if (ch == '{') {
         s.push(ch);
      }
      else if (ch == '}') {
         if (s.empty()) {
            balanced = false;
            break;
         }
         s.pop();
      }
   }

   if (!s.empty()) {
      balanced = false;
   }

   if (balanced)
      cout << "The string has balanced curly braces." << endl;
   else
      cout << "The string does NOT have balanced curly braces." << endl;

   return 0;
}
```

5.4 Time and Space Complexity

The time complexity of this program is O(n) because each character in the string is processed exactly once. The space complexity is also O(n) in the worst case, as the stack may store all opening braces.

5.5 Relation to Compiler Design

This problem is closely related to syntax analysis in compiler design. Compilers use stacks to check matching parentheses, braces, and brackets. Balanced symbols are essential for constructing valid parse trees, and this concept is widely used in parsers and syntax checkers.

6. (Problem-Solving): Parse Tree Construction

Given Grammar and String

The given grammar is $S \rightarrow aSb \mid \varepsilon$, and the given string is aabb.

6.1 Understanding the Grammar

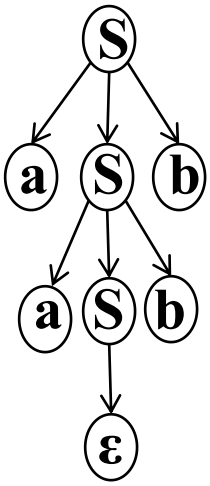
In this grammar, S is the start symbol, a and b are terminal symbols, and ε represents the empty string. This grammar generates strings in which the number of a symbols is equal to the number of b symbols, and all a symbols appear before the b symbols. Examples of strings generated by this grammar include ε, ab, aabb, and aaabbb.

6.2 Derivation of the String "aabb"

To construct the parse tree, we first derive the string step by step from the grammar. Starting with the start symbol S, we apply the production rule $S \rightarrow aSb$ to obtain a S b. Applying the same rule again results in a a S b b. Finally, replacing S with ε gives a a ε b b, which simplifies to aabb. This confirms that the string aabb is generated by the given grammar.

6.3 Parse Tree for the String "aabb"

A parse tree visually represents how a string is derived from a grammar. The root of the tree is the start symbol S, the internal nodes are non-terminals, and the leaf nodes are terminals or ε.



6.4 Explanation of the Parse Tree

The root node of the parse tree is S. The first expansion applies the rule S → aSb, followed by a second application of the same rule. The final S is replaced by ε. When the leaf nodes are read from left to right, they form the string a a b b, which is the string aabb.

Conclusion

In this assignment, the fundamental concepts of syntax analysis in compiler design were studied in detail. The discussion began with an explanation of syntax analysis and its role in verifying whether a sequence of tokens follows the grammatical rules of a programming language. Two major parsing techniques, Top-Down Parsing and Bottom-Up Parsing, were examined, highlighting their definitions, working principles, examples, advantages, and limitations. The comparison between these two approaches clearly showed how they differ in parse tree construction, derivation methods, grammar handling capability, and practical usage in real compilers.

The assignment also demonstrated a practical application of syntax analysis through a C++ program that checks for balanced curly braces using a stack. This example illustrated how parsing concepts are applied in real programs to validate structure and ensure correctness. Additionally, the parse tree construction problem provided a deeper understanding of how a grammar generates a string and how derivations are visually represented using parse trees.

Overall, this assignment helped strengthen the understanding of how compilers analyze program structure, detect errors, and ensure syntactic correctness. By combining theoretical concepts with practical examples, it clearly showed the importance of parsing techniques and syntax analysis in building reliable and efficient compilers.