
# Priniciple of Compiler Designer
## Individual Assignment
### Assignment 1:
### NO:57

Helen Adugna:
Id:1506501
Section:B

Subission Date:27/04/2018 ec
Submitted to:Wondmu Baye

## 1. (Theory): Define left recursion and explain why it is problematic in parsing.

Left recursion is a fundamental concept in grammar design for programming languages and plays a significant role in syntax analysis. A grammar is said to be left recursive if a non-terminal symbol can derive a sentential form in which the same non-terminal appears as the leftmost symbol. In simple terms, left recursion occurs when a grammar rule refers to itself before producing any terminal symbols.

Formally, a grammar is left recursive if there exists a non-terminal A such that:

$$A \Rightarrow^{+} A\alpha$$

where $\alpha$ represents a sequence of terminals and/or non-terminals, and $\Rightarrow^{+}$ denotes one or more derivation steps. Left recursion can occur either directly or indirectly. Direct (immediate) left recursion happens when a non-terminal appears directly on the right-hand side of its own production, such as:

$$E \rightarrow E + T$$

Indirect left recursion occurs when the recursion is introduced through intermediate non-terminals. Although less obvious, indirect left recursion can still cause serious issues during parsing.

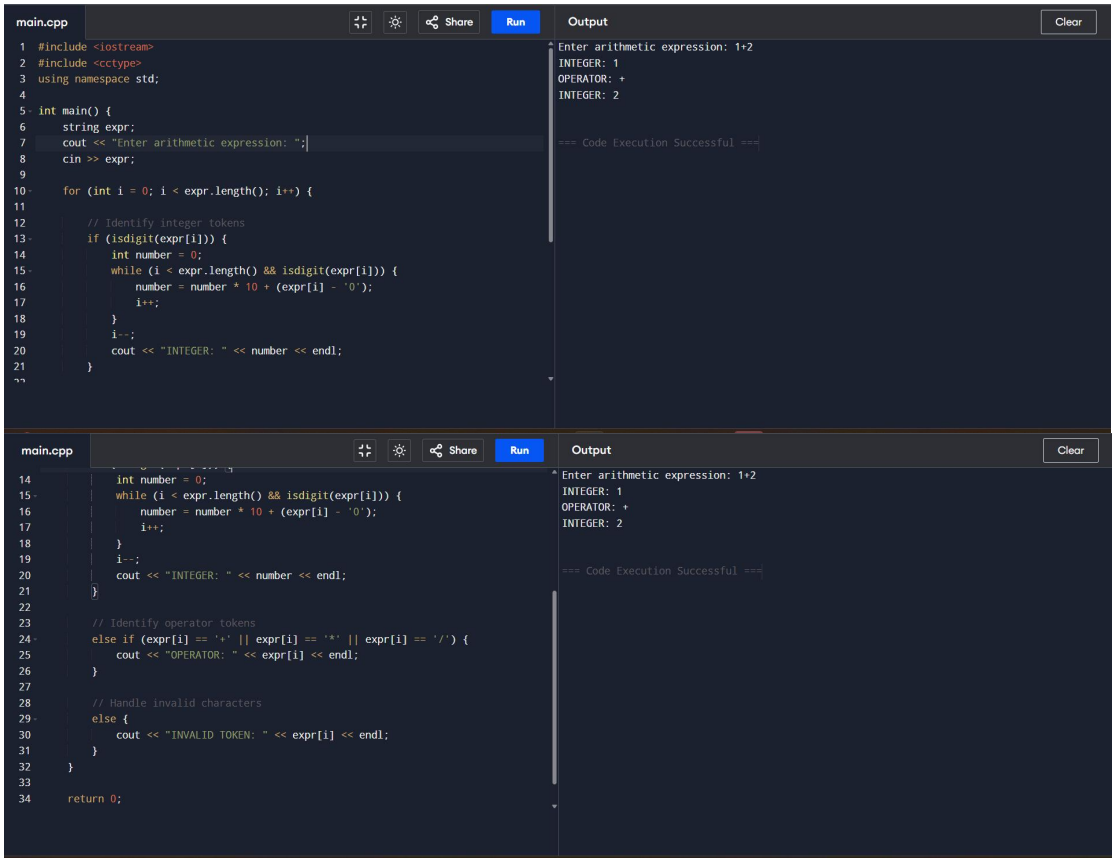
Left recursion becomes problematic mainly when using top-down parsing techniques, such as recursive descent parsers or LL parsers. These parsers attempt to expand productions starting from the start symbol and proceed by recursively applying grammar rules. When left recursion exists, the parser repeatedly expands the same non-terminal without consuming any input symbols. As a result, the parser enters an infinite loop and fails to terminate.

This behavior leads to several practical issues, including infinite recursion, excessive memory consumption, stack overflow errors, and parser failure. Because of these limitations, grammars containing left recursion are not suitable for predictive parsing methods like LL(1). Therefore, left recursion must be eliminated or transformed into an equivalent right-recursive form before applying top-down parsing techniques. Although bottom-up parsers such as LR parsers can handle left-recursive grammars, removing left recursion is still considered good practice for improving grammar clarity and parser reliability.

## 2. C++ Program: Tokenizing Arithmetic Expressions

Tokenization is a key activity performed during the lexical analysis phase of a compiler. The goal of tokenization is to convert a stream of input characters into a sequence of meaningful units known as tokens. In this task, the objective is to write a C++ program that identifies integer values and arithmetic operators (+, *, and /) from a given arithmetic expression.

The program works by scanning the input expression character by character. When a digit is encountered, the program continues reading consecutive digits to form a complete integer token. Arithmetic operators are recognized as individual tokens, while any unrecognized characters are reported as invalid tokens. This approach helps separate lexical analysis from syntax analysis, making the overall compiler structure more modular and easier to manage.




This program clearly demonstrates how lexical scanning is performed. By separating numbers and operators into tokens, it prepares the input for the next phase of compilation, which is syntax analysis

## 3. Problem-Solving: Parse Tree Construction

Given Grammar:
S → AB
A → aA | ε
B → bB | b

Input String: aab

## Grammar Interpretation

In this grammar, the non-terminal A generates zero or more occurrences of the terminal symbol ‘a’, while the non-terminal B generates one or more occurrences of the terminal symbol ‘b’. The start symbol S combines both parts by expanding into AB. This structure ensures that any valid string produced by the grammar consists of a sequence of 'a' characters followed by at least one 'b'.

Step-by-Step Derivation

The derivation process begins with the start symbol:

S → AB

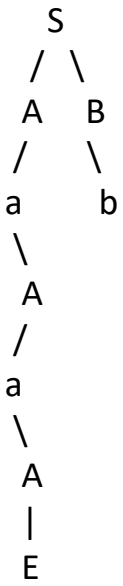
To generate the prefix "aa", non-terminal A is expanded twice using the recursive rule:
A → aA
A → aA

The recursion is then terminated using the epsilon production:
A → ε

Next, non-terminal B is expanded to produce the final character:
B → b

Complete Derivation:
S → AB
→ aA B
→ aaA B
→ aa ε B
→ aab

Parse Tree Representation




## Explanation of the Parse Tree

The root of the parse tree is the start symbol S, which expands into A and B. The recursive expansions of A generate two 'a' terminals, while the epsilon (ε) production signals the end of recursion. The non-terminal B generates a single 'b'. Reading the leaf nodes from left to right produces the string "aab", confirming that the string is valid according to the grammar.

Parse trees are essential in syntax analysis because they visually represent the grammatical structure of a string. They help verify correctness, understand derivations, and serve as the basis for constructing abstract syntax trees (ASTs), which are used in later compiler phases such as semantic analysis and code generation.

## Conclusion

This assignment provided a comprehensive exploration of syntax analysis, highlighting its theoretical foundations, practical implementation, and problem-solving aspects within compiler design. Through the discussion of left recursion, lexical tokenization using C++, and parse tree construction, the assignment successfully demonstrated how grammatical rules are defined, analyzed, and applied during the compilation process.

The theoretical analysis of left recursion emphasized its significance in grammar design and parser construction. By understanding how left recursion arises and why it causes infinite recursion in top-down parsers, the importance of grammar transformation becomes clear. This concept illustrates that not all grammars are immediately suitable for all parsing techniques, and careful analysis is required to ensure parser compatibility, correctness, and efficiency. Recognizing and eliminating left recursion is therefore a critical skill for designing reliable syntax analyzers.

The implementation of a C++ tokenizer for arithmetic expressions translated theoretical compiler concepts into practical programming tasks. By identifying integers and arithmetic operators from an input expression, the program demonstrated the role of lexical analysis as the foundation of syntax analysis. This implementation showed how raw input characters are converted into structured tokens that can be easily processed by parsers. It also reinforced the importance of separating lexical analysis from syntax analysis to simplify compiler design and improve modularity.
The parse tree construction problem further strengthened understanding by applying grammar rules to a real input string. Constructing the parse tree for the string “aab” clearly illustrated how recursive productions and epsilon rules operate within a context-free grammar. This exercise highlighted the hierarchical structure of syntactic derivations and showed how parse trees represent the complete grammatical structure of a string. Such representations are essential for validating syntax, detecting errors, and forming the basis for abstract syntax trees used in later compiler phases.

Overall, this assignment effectively integrated theory, implementation, and analytical reasoning to build a strong foundation in syntax analysis. The concepts and skills developed here are essential for advanced topics such as semantic analysis, code generation, and compiler optimization. By combining grammar theory with hands-on C++ programming and structured problem solving, the assignment enhanced both conceptual understanding and practical competence in compiler design.