# SyntaxAnalyzer Documentation

Design and Implementation of Syntax Analyzer

Tyler LeBlanc

Introduction to Compiling

Professor Fatima

Wilfrid Laurier University

07/03/2025

## Introduction

The syntax analyzer is the second phase of the compiler. Its primary role is to validate that the sequence of tokens produced by the lexical analyzer conforms to the language's grammar. This phase ensures that the source program is syntactically correct, setting the stage for subsequent compilation processes.

## Overview of Syntax Analysis

Syntax analysis, commonly known as parsing, involves checking the token stream against a formal grammar. By using a predictive parsing approach—specifically an LL(1) parser—the syntax analyzer efficiently determines whether the program's structure adheres to the defined grammar rules.

## Design of the Syntax Analyzer

The syntax analyzer is based on the LL(1) parsing methodology. This design includes the following key aspects:

- **LL(1) Parsing Table:**
  The parser employs a precomputed LL(1) parsing table that maps nonterminal symbols and lookahead terminal symbols to specific productions. Each entry in this table represents a production rule to be applied during parsing.
- **Stack-Based Parsing:**
  A stack is used to keep track of expected symbols. Initially, the start symbol of the grammar is pushed onto the stack. The parser then iterates through the tokens produced by the lexical analyzer, comparing the top of the stack with the current token:
  - If the top of the stack is a terminal and it matches the current token, the symbol is popped from the stack.
  - If the top is a nonterminal, the parser consults the LL(1) table to determine the appropriate production rule and pushes the symbols of that production (in reverse order) onto the stack.
  - This process continues until the stack is empty and the entire token stream is consumed.

- **Error Handling:**
  When a token does not match the expected terminal, or no valid production is found for a nonterminal/token pair, the parser reports a syntax error.

## Major Components

- **Main Program (main.c):**
  Integrates both lexical and syntax analysis. After tokenization, it passes the token stream to the syntax analyzer to verify the program's grammatical structure.

- **SyntaxAnalysis.h:**
  Contains declarations for key types and functions used in syntax analysis. Major structures include the stack for parsing, enumerations for terminal and nonterminal symbols, and the LL(1) parsing table.

## Key Structures and Definitions

- **Terminal and Nonterminal Enumerations:**
  These enumerations define all the terminal symbols (such as keywords, operators, and punctuation) and nonterminal symbols (representing syntactic categories) used in the grammar.

- **Parsing Table Entry:**
  Each entry in the LL(1) table is associated with a production rule represented as a string. This table guides the parser in expanding nonterminals based on the current input token.

- **Stack Structure:**
  A stack is implemented to manage the symbols during parsing. The push, pop, and peek operations facilitate the comparison of expected symbols against the input token stream.

## Algorithm Explanation

- **Initialization:**
  The parser initializes the stack with the start symbol of the grammar. It also prepares to read the sequence of tokens generated by the lexical analyzer.
- **Parsing Loop:**
  The parser reads the current token from the token stream. It examines the symbol on the top of the stack:
  - **Terminal Match:** If the symbol is a terminal and matches the current token, the symbol is popped from the stack, and the parser advances to the next token.
  - **Nonterminal Expansion:** If the symbol is a nonterminal, the parser consults the LL(1) parsing table. If a valid production exists for the current token, the nonterminal is replaced (popped) and the right-hand side of the production is pushed onto the stack in reverse order.
  - **Error Detection:** If neither case applies or no valid production rule is found, the parser detects a syntax error.
- **Termination:**
  The parsing process continues until the stack is empty and all tokens have been processed. Successful completion indicates that the input conforms to the grammar.

## Handling Syntax Errors

When the parser encounters an unexpected token or an invalid production rule:

- **Error Reporting:**
  The parser outputs an error message indicating the nature of the syntax error, including information such as the unexpected token and its location.
- **Recovery Strategy:**
  Once an error is reported, the parser skips the offending token and resumes processing the stream.

## Challenges and Limitations

- **Challenges:**
  Properly implementing the LL(1) table to properly guide the parser to the correct terminals was a big issue in the beginning of our implementation.

- **Limitations:**
  Implementing robust error recovery without compromising the parser's correctness is challenging. The balance between informative error reporting and graceful recovery remains a key design consideration.

## Conclusion

The syntax analyzer successfully validates the grammatical structure of the input token stream using an LL(1) parsing approach. By leveraging a parsing table and a stack-based algorithm, it ensures that the source program adheres to the defined language grammar.