Ministry of Education, Culture, and Research of the Republic of MoldovaTechnical University of Moldova
The Faculty of Computers, Informatics, and Microelectronics

REPORT

Laboratory work no.6
Parser & Building an Abstract Syntax Tree

Executed by: Cornievschi Bogdan
gr. FAF-222

Verified by:
univ. assist. Crețu Dumitru

Chişinău - 2024

Introduction

Following lexical analysis, syntactic analysis (parsing) takes place. The parser takes the linear sequence of tokens produced by the lexer and constructs a hierarchical structure known as an abstract syntax tree (AST). The AST represents the grammatical structure of the input code, organizing tokens into a tree that reflects their syntactic relationships and the rules of the programming language's grammar. This structure is crucial for further steps in the compilation or interpretation process, such as semantic analysis, optimization, and code generation.

The objective of this lab is to design and implement a simple lexer and parser for a hypothetical programming language. The lexer will identify tokens such as integers, floats, identifiers, keywords, and operators from a given input string. The parser will then interpret these tokens to build an AST, which can represent arithmetic operations, variable declarations, and control flow constructs like conditional statements. Through this exercise, students will gain hands-on experience with the mechanisms underlying the initial phases of compiling and understand how high-level constructs are translated into structured representations that are essential for creating executable programs.

Implementation of the Lexer

The lexer, also known as a lexical analyzer or scanner, is the first phase of the compiler that processes the input code. In this lab, the lexer was implemented to scan a predefined set of characters from an input string and convert them into a meaningful sequence of tokens. Each token represents a fundamental element of the language, such as a datatype, identifier, operator, or control structure.

Functionality:

- Token Recognition: The lexer recognizes categories including integers, floating-point numbers, identifiers, keywords (such as if, else, and return), operators (+, -, *, /, >, <), and punctuation marks (;, (,)). It handles the skipping of whitespace to ensure that only meaningful characters are processed.
- Error Handling: The lexer also includes basic error handling to report unrecognized characters, enhancing robustness and user feedback.

Methodology:

- The process begins with the lexer reading the input character by character. Depending on the character, it decides whether to group it into a multi-character token, identify it as a standalone token, or discard it as whitespace.
- For numeric tokens, the lexer distinguishes between integers and floats. This is achieved by detecting a decimal point, which signifies a float. Otherwise, the number is categorized as an integer.
- Identifiers and keywords are distinguished by checking against a list of reserved words; if a sequence of characters matches a reserved word, it is labeled as a keyword, else it is treated as an identifier.

Implementation of the Parser

The parser is responsible for taking the stream of tokens generated by the lexer and assembling them into an abstract syntax tree (AST) that represents the syntactic structure of the input code. This process involves recognizing patterns in the sequence of tokens that conform to the grammar of the language.

Functionality:

- Constructing the AST: The parser we implemented follows a recursive descent parsing technique, which involves a series of functions that mirror the grammar of the language. It handles arithmetic expressions, variable assignments, and control flow constructs.
- Error Handling: Similar to the lexer, the parser includes error handling to manage and report syntactic errors by throwing exceptions when the token sequences do not match the expected patterns.

Methodology:

- Starting from the highest level of the grammar (such as statements), the parser recursively calls functions that represent non-terminal symbols in the grammar. These functions move through the token list and construct nodes of the AST.
- Each function corresponds to a specific grammatical rule, and the AST nodes are created accordingly. For instance, a binary operation like addition (a + b) results in a BinaryOpNode with children representing the operands.

Structure of the Abstract Syntax Tree (AST)

The AST is a pivotal data structure in many compilers and interpreters. It represents the syntactic structure of the parsed code in a tree-like form, which simplifies further processing such as analysis, optimisation, and code generation.

Components:

- Nodes: Each node in the AST corresponds to a construct in the language. For example, BinaryOpNode for binary operations, NumNode for numeric literals, and IfNode for conditional statements.
- Leaf and Internal Nodes: Leaf nodes typically represent literals or identifiers, while internal nodes represent expressions or control structures.

Advantages:

- Clarity and Simplicity: The tree structure clearly separates different parts of the syntax, making it easier to apply further transformations and optimisations.
- Facilitation of Static Analysis: The AST allows for efficient traversal and manipulation during static analysis, type checking, and other compiler phases.

Input / Output example

Input: ”if (x > 50) return x * 10;”

Output:

```
Consuming: Token(TokenType.KEYWORD, 'if')
Consuming: Token(TokenType.LPAREN, '(')
Consuming: Token(TokenType.IDENTIFIER, 'x')
Consuming: Token(TokenType.OPERATOR, '>')
Consuming: Token(TokenType.INTEGER, 50)
Consuming: Token(TokenType.RPAREN, ')')
Consuming: Token(TokenType.KEYWORD, 'return')
Consuming: Token(TokenType.IDENTIFIER, 'x')
Consuming: Token(TokenType.OPERATOR, '*')
Consuming: Token(TokenType.INTEGER, 10)
Consuming: Token(TokenType.SEMICOLON, ';')
IfNode(BinaryOpNode(VarNode(x), >, NumNode(50)),
        ReturnNode(BinaryOpNode(VarNode(x), *, NumNode(10))), None)
```

Conclusion

The successful completion of this laboratory exercise has provided invaluable insights into the fundamental processes of lexical and syntactic analysis in the compilation of programming languages. By implementing a lexer and a parser, and integrating these components to construct an Abstract Syntax Tree (AST), we have demonstrated a basic yet powerful framework for understanding how high-level programming languages are transformed into structured formats that a machine can begin to execute.

- **Lexer Implementation**: The development of the lexer enabled us to effectively translate a raw input string into a sequence of meaningful tokens. This process not only emphasized the importance of recognizing and categorizing language elements but also highlighted the necessity for careful error handling to manage unexpected characters.
- **Parser Development**: Building the parser taught us how to take a flat sequence of tokens and organize them into a hierarchical structure that reflects the program's syntax. This step was crucial for representing nested constructions and logical sequences in the code, such as expressions and control flows.
- **AST Construction**: The construction of the AST from tokens facilitated a deeper understanding of the program's syntactic structure, proving essential for subsequent phases of a compiler, such as semantic analysis, optimization, and code generation.