Phase 2: Syntax Analysis – Parser Implementation

CISC 458

Shams Sajid - 20217791

Harshil Patel - 20250563

Cj Gonsalves - 20230353

Devin Chawla - 20223139

# Overview

In Phase 2, a parser has been implemented to extend the previously created lexical analyzer. The parser constructs an Abstract Syntax Tree (AST) from the provided token stream, validating the syntactic structure of input according to the defined language grammar.

## AST Structure

The AST is a hierarchical, tree-like data structure that clearly reflects the syntactic relationships within the parsed source code. Each node in the AST represents a syntactic construct, such as variables, assignments, expressions, or control structures, and stores relevant token information along with pointers to child nodes. This clear structure aids in semantic analysis and subsequent code generation stages.

## Expression Parsing

The given code's expression parsing handle mathematical expressions methodically, handling numerous operators to ensure correct precedence and associativity. The parsing processes identify and evaluate operators including addition (+), subtraction (-), multiplication (\*), division (/), exponentiation (^), and modulus (%). The parser first evaluates primary values such as numbers or grouped expressions within parentheses before applying operators according to their precedence levels. Multiplication and division take precedence over addition and subtraction, while exponentiation is given the highest priority. The recursive parsing approach creates an Abstract Syntax Tree (AST) with each node representing an operation, guaranteeing that equations are processed appropriately using operator precedence rules.

An example of expression parsing in the code would be parse\_term() which handles addition and subtraction.

A screen shot of a computer

Description automatically generated

## Statement Parsing

Supported parsing of statements:

* Variable declarations (int x;).
* Assignments (x = expression;).
* Control structures including:
  + if statements (if (condition) {statements}).
  + while loops (while (condition) {statements}).
  + repeat-until loops (repeat {statements} until (condition);).
* print statements (print expression;).
* Block statements ({statement1; statement2;}).

Parsing statements involves identifying the initial keyword or token and branching into specialized parsing functions. For example, an if statement is parsed by first consuming the 'if' keyword, then parsing the condition within parentheses, followed by parsing the statement block within braces.

**Example:**

**A computer code with text

AI-generated content may be incorrect.**

Figure 1: Example of the statement parsing

## Special Features

This parser implementation includes several notable features: **factorial function support**, **runtime error detection**, and **block scoping**. The factorial function is handled as a function call (parse\_factorial), allowing users to compute factorial values within expressions. Runtime error detection is implemented through parse\_error, which provides detailed messages for syntax errors such as unexpected tokens, missing semicolons, and invalid expressions, improving debugging. Block scoping is managed by parse\_block, ensuring that statements enclosed in {} form a structured execution unit, preventing unintended variable leaks. These features enhance the parser’s robustness and usability.

A screen shot of a computer code

Description automatically generated

Figure - Function for factorial function support

A screen shot of a computer code

Description automatically generated

Figure - Function for block scoping

## Error Handling

Robust error handling is crucial for clear diagnostics and graceful failure in syntax parsing. Errors are precisely reported with line numbers and informative messages, aiding debugging and improving user experience. Enhanced error reporting with clear messages and specific error types (e.g., PARSE\_ERROR\_MISSING\_SEMICOLON, PARSE\_ERROR\_UNEXPECTED\_TOKEN).

**Example:**

In this example, the ‘parse\_error’ function outputs detailed information about encountered syntax issues. When an unexpected token is detected, it specifies the exact token and its location, enhancing user understanding of errors and simplifying debugging.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 5: Example of the parse error handling

## Testing and Debugging

Testing and debugging were conducted using both valid and invalid input cases.

1. The **invalid input test** demonstrated the parser's ability to identify and report syntax errors accurately, as seen when the parser reported a "Missing block braces after 'EOF'" error clearly pinpointing the exact line causing the issue.
2. The **valid input AST test** case shows the Abstract Syntax Tree (AST) generated from valid input. It includes variable declarations (VarDecl: x), assignments, control structures (If, While, Repeat-Until), and expressions.
3. The **valid input test** case shows the original valid source code input tested containing diverse statement types including declarations, assignments, conditionals, loops, and function calls.

A screen shot of a computer

AI-generated content may be incorrect.

Figure : image test case valid input

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 7: Test case Valid Input AST - Test case valid input image

A screenshot of a computer program

AI-generated content may be incorrect.

Figure : Test case invalid input image

## Conclusion

The implementation of the parser is a significant step to creating a fully functional compiler. By effectively converting a tokenized input stream into an Abstract Syntax Tree (AST), the parser ensures proper syntactic structuring, operator precedence, and statement and expression handling. Advanced features like factorial function support, block scoping, and runtime error detection make the parser more robust and usable. The error-handling mechanisms enable accurate diagnostics, which improves debugging and user experience. Comprehensive testing, using both valid and invalid inputs, demonstrated the parser's ability to correctly construct the AST and gracefully handle syntax errors. With a solid foundation in place, this parser is ready to support future compiler development, such as semantic analysis and code generation.