

CS424

Assignment 2 - REPORT



Ghulam Ishaq Khan Institute of Science and Technology

REG#: 2020447

Name: Shahryar Mubashar
INSTRUCTOR: USAMA ARSHAD

Introduction:

The objective of this report is to provide a comprehensive overview of the parser implementation for MiniLang, focusing on key aspects such as parser type selection, grammar rules definition, syntax tree construction, and error handling mechanisms. By understanding the intricacies of parser design and implementation, readers will gain insights into the core principles of compiler construction and be equipped with the knowledge to tackle similar challenges in their own projects.

Parser Implementation:

Parser Type:

For the MiniLang programming language, a top-down parser, specifically a recursive descent parser, is chosen for implementation. This decision is based on the simplicity of MiniLang and the ease of implementing a recursive descent parser for languages with straightforward grammars. Recursive descent parsers are intuitive and align well with the structure of MiniLang, making them suitable for parsing this language efficiently.

Grammar Rules:

The grammar rules for MiniLang are defined to cover essential language constructs such as expressions, statements, and programs. Each rule corresponds to a specific construct in MiniLang, facilitating the parsing process and ensuring the parser can accurately recognize and validate the syntax of MiniLang programs. The grammar rules are designed with clarity and conciseness in mind, allowing for easy implementation and maintenance.

Syntax Tree:

The parser builds an abstract syntax tree (AST) that represents the hierarchical structure of the MiniLang source code. Each node in the tree corresponds to a language construct, such as expressions, statements, or program elements. By constructing an AST, the parser organizes the parsed code in a structured manner, facilitating subsequent stages of the compiler pipeline, such as semantic analysis and code generation.

Error Handling:

Error handling mechanisms are implemented in the parser to report syntax errors encountered during the parsing process. These mechanisms provide meaningful error messages that help developers identify and rectify syntax errors in their MiniLang programs.

Scanner Design and Implementation:

Design Decisions:

Tokenization Approach: Utilized a simple approach where the scanner iterates through the input text character by character to identify tokens.

Token Types: Defined token types based on the MiniLang language specifications, including integers, arithmetic operators, parentheses, etc.

Error Handling: Implemented error handling to raise an exception for invalid characters encountered during tokenization.

Structure of the Scanner:

Lexer Class: Handles the tokenization process by iterating through the input text and generating tokens.

Token Class: Represents individual tokens with attributes such as token type and value.

How to Run the Scanner:

Initialize a Lexer object with the input text.

Call the `get_next_token()` method of the Lexer to obtain tokens iteratively.

Test Cases:

1. Simple Arithmetic Expression:

Input: "2 + 3 * 4"

Expected Output: [INTEGER(2), PLUS, INTEGER(3), MULTIPLY, INTEGER(4)]

```
def main():
    while True:
        try:
            text = input("MiniLang> ")
        except EOFError:
            break
        if not text:
            continue
        lexer = Lexer(text)
        parser = Parser(lexer)
        tree = parser.parse()
        print_ast(tree)

if __name__ == "__main__":
    main()
```

```
MiniLang> 2 + 3 * 4
PLUS
2
MULTIPLY
3
4
MiniLang> 
```

2. Complex Expression with Parentheses:

Input: "(5 + 2) * (3 - 1)"

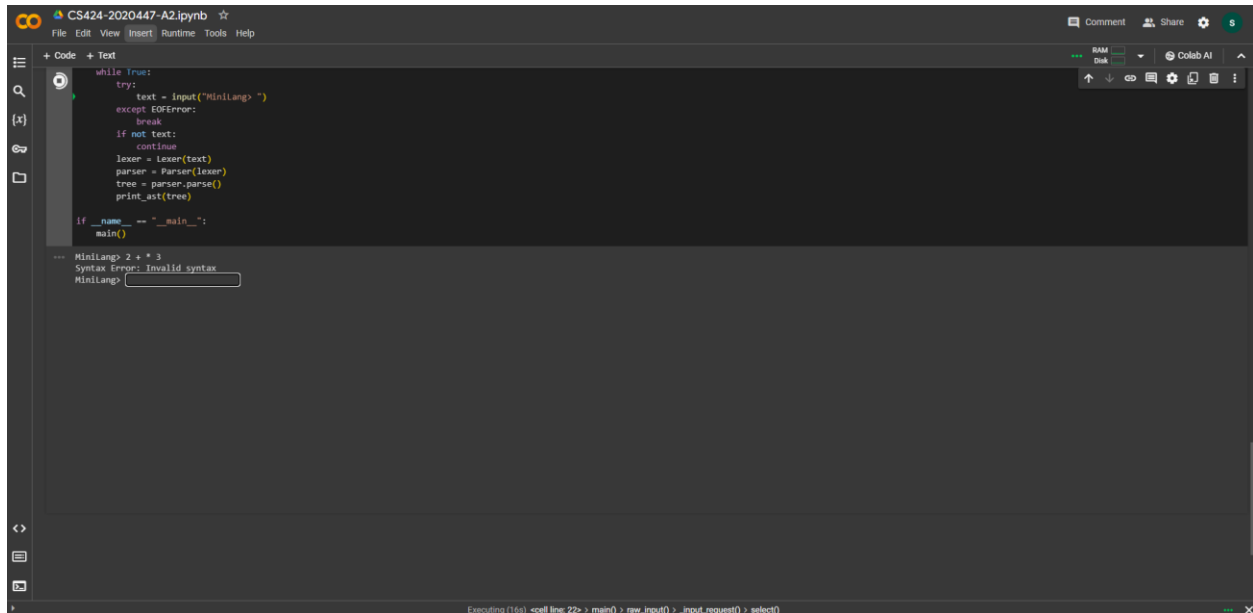
Expected Output: [LPAREN, INTEGER(5), PLUS, INTEGER(2), RPAREN, MULTIPLY, LPAREN, INTEGER(3), MINUS, INTEGER(1), RPAREN]

```
MiniLang> (5 + 2) * (3 - 1)
MULTIPLY
PLUS
5
2
MINUS
3
1
MiniLang> 
```

3. Error Handling - Invalid Character:

Input: "2 + * 3"

Expected Output: Exception raised for encountering * as an unexpected character.



The screenshot shows a Jupyter Notebook interface with a code cell containing the following Python code:

```
while True:
    try:
        text = input("MiniLang> ")
    except EOFError:
        break
    if not text:
        continue
    lexer = Lexer(text)
    parser = Parser(lexer)
    tree = parser.parse()
    print_ast(tree)

if __name__ == "__main__":
    main()
```

Below the code cell, the notebook shows the execution of the code with the input "2 + * 3". The output is:

```
MiniLang> 2 + * 3
Syntax Error: Invalid syntax
MiniLang>
```

The status bar at the bottom indicates the execution is running on line 22, column 22, in the `main()` function, with the error message `Syntax Error: Invalid syntax`.

Conclusion:

The MiniLang scanner efficiently tokenizes input text according to the language specifications, providing essential functionality for further stages of the compiler. With error handling and support for various input scenarios, it ensures robustness and reliability in parsing MiniLang programs.