Laboratory work nr.6
Parser & Building an Abstract Syntax Tree

Course: Formal Languages and Automana
Student: Schipschi Daniil FAF-223

Objectives:

Overview

The process of gathering syntactical meaning or doing a syntactical analysis over some text can also be called parsing. It usually results in a parse tree which can also contain semantic information that could be used in subsequent stages of compilation, for example.

Similarly to a parse tree, in order to represent the structure of an input text one could create an Abstract Syntax Tree (AST). This is a data structure that is organized hierarchically in abstraction layers that represent the constructs or entities that form up the initial text. These can come in handy also in the analysis of programs or some processes involved in compilation.

Objectives:

Get familiar with parsing, what it is and how it can be programmed [1].

Get familiar with the concept of AST [2].

In addition to what has been done in the 3rd lab work do the following:

In case you didn't have a type that denotes the possible types of tokens you need to:

Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens.

Please use regular expressions to identify the type of the token.

Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.

Implement a simple parser program that could extract the syntactic information from the input text.

Implementation:

```
package ast_parser;

public enum TokenType {
    NUMBER,
    OPERATOR,
    LPAREN,
    RPAREN,
    FUNCTION,
    COMMA,
    EOF
}
```

In the code snippet, I defined an enumeration called TokenType in the ast_parser package, which categorizes different types of tokens needed for parsing expressions:

NUMBER: Represents numeric values.
OPERATOR: Represents arithmetic operators like +, -, *, and /.
LPAREN and RPAREN: Represent left (and right) parentheses for managing operation precedence.
FUNCTION: Represents function names such as sqrt or log.
COMMA: Used for separating function arguments.
EOF: Marks the end of input to signify completion of parsing.

```java
package ast_parser;

public class Token {
    public TokenType type;
    public String value;

    public Token(TokenType type, String value) {
        this.type = type;
        this.value = value;
    }

    @Override
    public String toString() {
        return type + " : " + value;
    }
}
```

Represents tokens with a type (like NUMBER or OPERATOR) and a value, used during the parsing of expressions.

```java
package ast_parser;

abstract class Expr {
    abstract double evaluate();
}
```

Serves as the base for all expression types, requiring subclasses to implement a method to evaluate and return a numeric result.

```java
package ast_parser;

class NumberExpr extends Expr {
    double value;

    NumberExpr(double value) {
        this.value = value;
    }

    @Override
    double evaluate() {
        return value;
    }
}
```

Handles numeric values within expressions, directly returning the number when evaluated.

```java
package ast_parser;

import java.util.List;

class FunctionExpr extends Expr {
    String function;
    List<Expr> arguments;

    FunctionExpr(String function, List<Expr> arguments) {
        this.function = function;
        this.arguments = arguments;
    }

    @Override
    double evaluate() {
        return switch (function) {
            case "sqrt" -> {
                if (arguments.size() != 1) {
                    throw new RuntimeException("sqrt expects exactly one argument");
                }
                yield Math.sqrt(arguments.get(0).evaluate());
            }
            case "logtwo" -> {
                if (arguments.size() != 1) {
                    throw new RuntimeException("log2 expects exactly one argument");
                }
                yield Math.log(arguments.get(0).evaluate()) / Math.log(2);
            }
            case "logten" -> {
                if (arguments.size() != 1) {
                    throw new RuntimeException("log10 expects exactly one argument");
                }
                yield Math.log10(arguments.get(0).evaluate());
            }
            case "loge" -> {
                if (arguments.size() != 1) {
                    throw new RuntimeException("loge expects exactly one argument");
                }
                yield Math.log(arguments.get(0).evaluate());
            }
            case "powtwo" -> {
                if (arguments.size() != 1) {
                    throw new RuntimeException("pow2 expects exactly one argument");
                }
                yield Math.pow(2, arguments.get(0).evaluate());
            }
            default -> throw new RuntimeException("Unsupported function or wrong number of
arguments: " + function);
        };
    }
}
```

Manages function calls in expressions, evaluating various mathematical functions based on the function name and arguments provided.

```java
package ast_parser;

class BinaryOperatorExpr extends Expr {
    Expr left;
    Expr right;
    char operator;

    BinaryOperatorExpr(Expr left, Expr right, char operator) {
        this.left = left;
        this.right = right;
        this.operator = operator;
    }

    @Override
    double evaluate() {
        switch (operator) {
            case '+': return left.evaluate() + right.evaluate();
            case '-': return left.evaluate() - right.evaluate();
            case '*': return left.evaluate() * right.evaluate();
            case '/': return left.evaluate() / right.evaluate();
            default: throw new RuntimeException("Unsupported operator: " + operator);
        }
    }
}
```

The BinaryOperatorExpr class handles basic arithmetic operations involving two operands. It uses an operator (like +, -, *, /) to combine the results of evaluating its two operand expressions (left and right). This allows it to compute and return the result of the operation.

```java
package ast_parser;

import java.util.ArrayList;
import java.util.List;

public class Parser {
    private final List<Token> tokens;
    private int current = 0;

    public Parser(List<Token> tokens) {
        this.tokens = tokens;
    }

    public Expr parse() {
        return expression();
    }

    private Expr expression() {
        Expr expr = term();
        while (match(TokenType.OPERATOR)) {
            char operator = tokens.get(current - 1).value.charAt(0);
            Expr right = term();
            expr = new BinaryOperatorExpr(expr, right, operator);
        }
        return expr;
    }

    private Expr term() {
        if (match(TokenType.NUMBER)) {
            return new NumberExpr(Double.parseDouble(previous().value));
        } else if (match(TokenType.LPAREN)) {
            Expr expr = expression();
            consume(TokenType.RPAREN, "Expect ')' after expression.");
            return expr;
        } else if (match(TokenType.FUNCTION)) {
            String functionName = previous().value;
            consume(TokenType.LPAREN, "Expect '(' after function name.");
            List<Expr> args = new ArrayList<>();
            if (!check(TokenType.RPAREN)) {
                do {
                    args.add(expression());
                } while (match(TokenType.COMMA));
            }
            consume(TokenType.RPAREN, "Expect ')' after arguments.");
            return new FunctionExpr(functionName, args);
        }
        throw new RuntimeException("Unexpected token.");
    }

    private boolean match(TokenType type) {
        if (check(type)) {
            current++;
            return true;
        }
        return false;
    }

    private boolean check(TokenType type) {
        if (isAtEnd()) return false;
        return tokens.get(current).type == type;
    }

    private Token previous() {
        return tokens.get(current - 1);
    }

    private void consume(TokenType type, String message) {
        if (check(type)) {
            current++;
            return;
        }
        throw new RuntimeException(message);
    }

    private boolean isAtEnd() {
        return current >= tokens.size() || tokens.get(current).type == TokenType.EOF;
    }
}
```

The Parser class in the ast_parser package is designed to convert a sequence of tokens into an abstract syntax tree (AST) that represents the mathematical expressions those tokens encode. Here's how it functions:

Initialization: The Parser is initialized with a list of tokens which it processes sequentially.
Parsing Methodology:
parse(): Initiates parsing by calling the expression() method and returns the resulting Expr AST node, which represents the entire expression.
expression(): Handles arithmetic operations that follow typical operator precedence. It constructs Expr nodes for binary operations, repeatedly applying operators to the terms parsed.
term(): Parses individual terms within the expression, handling numbers, parenthesized expressions, and functions.

It manages the nesting of expressions and the creation of function calls with their arguments.
Utility Methods:
match() and check(): Used to verify if the current token matches expected types, aiding in syntactical validation.
previous(), consume(), and isAtEnd(): Help manage the token list by advancing the token index and ensuring tokens are consumed in the correct order, or throwing errors when unexpected tokens are encountered.
The parser effectively builds a tree structure that captures the hierarchy and dependencies of operations and functions in the input expression. This structure can then be evaluated to compute the result of the expression.

Conclusion:
In this laboratory, I learned how to construct Abstract Syntax Trees (ASTs) and parse mathematical expressions, which helped me understand the fundamentals of lexical analysis, syntax parsing, and expression evaluation. By implementing classes such as Token, Expr, and Parser within the ast_parser package, I gained hands-on experience in defining and managing tokens, creating various expression types, and systematically converting a sequence of tokens into structured ASTs.

This process involved distinguishing between different types of tokens, handling operations and function calls, and dealing with operator precedence and parentheses in expressions. Through the practical application of creating a parser and defining expression evaluation methods, I've enhanced my understanding of how parsers and compilers work internally to interpret and evaluate input according to predefined rules.

Overall, this laboratory exercise has been instrumental in deepening my appreciation of the complexities involved in programming language design and interpretation, laying a strong foundation for further exploration of compilers and interpreters.