Building a Compiler for a Simple Calculator

Project Report

دانشگاه صنعتی خواجه نصیرالدین طوسی

Prof. Nasersharif
Fatemeh Amirabadi

Writing the Calculator Grammar

First, we need to define the grammar for our calculator language in Backus-Naur Form (BNF). This grammar supports addition, subtraction, multiplication, division, and parentheses for grouping.

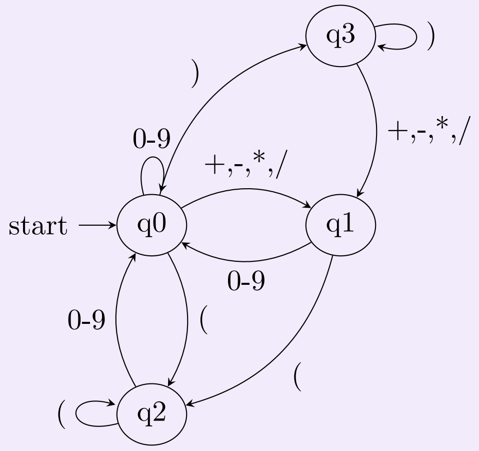
Explanation:

<expression> ::= <term> (("+" | "-") <term>)*
<term> ::= <factor> (("*" | "/") <factor>)*
<factor> ::= <number> | "(" <expression> ")"
<number> ::= [0-9]+

Designing the Calculator DFA

We should design a DFA that recognizes the language generated by this grammar. The DFA will need to account for the following tokens:[Numbers ([0-9]+) and Operators (+, -, *, /) and Parentheses ((,))]

Explanation:



Designing the Compiler

Class: Calculator

```java

public class Calculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.print("Enter expression: ");
            String input = scanner.nextLine();
            if (input.equalsIgnoreCase("exit")) {
                break;
            }
            try {
                Lexer lexer = new Lexer(input);
                List<Token> tokens = lexer.tokenize();
                Parser parser = new Parser(tokens);
                Node ast = parser.parse();
                Evaluator evaluator = new Evaluator();
                int result = evaluator.evaluate(ast);
                System.out.println("Result: " + result);
            } catch (Exception e) {
                System.out.println("Error: " + e.getMessage());
            }
        }
        scanner.close();
    }
}
```

Explanation:

The Calculator class contains the main method, which serves as the entry point of the program.

Prompts the user to enter an arithmetic expression.
Initializes a Lexer object with the input string.
Initializes a Parser object with the list of tokens.
Evaluates the AST to compute the result.

Class: Evaluator

```java
class Evaluator {
    int evaluate(Node node) {
        if (node instanceof NumberNode) {
            return ((NumberNode) node).value;
        } else if (node instanceof BinOpNode) {
            BinOpNode binOpNode = (BinOpNode) node;
            int left = evaluate(binOpNode.left);
            int right = evaluate(binOpNode.right);
            switch (binOpNode.op.type) {
                case PLUS:
                    return left + right;
                case MINUS:
                    return left - right;
                case MULTIPLY:
                    return left * right;
                case DIVIDE:
                    return left / right;
                default:
                    throw new RuntimeException("Unexpected operator: " +
                        binOpNode.op.type);
            }
        }
        throw new RuntimeException("Unexpected node type: " + node.getClass().
            getName());
    }
}
```

Explanation:

The Evaluator class is responsible for evaluating the AST and computing the result.
Returns the value if the node is a NumberNode.
Processes binary operation nodes.
Recursively evaluates the left and right subtree.

Class: Lexer

```java

class Lexer {
    private final String input;
    private int pos = 0;
    private char currentChar;

    Lexer(String input) {
        this.input = input;
        this.currentChar = input.charAt(pos);
    }

    private void advance() {
        pos++;
        if (pos >= input.length()) {
            currentChar = ’\0’;
        } else {
            currentChar = input.charAt(pos);
        }
    }

    private void skipWhitespace() {
        while (currentChar != ’\0’ && Character.isWhitespace(currentChar)) {
            advance();
        }
    }

    private String number() {
        StringBuilder result = new StringBuilder();
        while (currentChar != ’\0’ && Character.isDigit(currentChar)) {
            result.append(currentChar);
            advance();
        }
        return result.toString();
    }

    List<Token> tokenize() {
        List<Token> tokens = new ArrayList<>();
        while (currentChar != ’\0’) {
            if (Character.isWhitespace(currentChar)) {
                skipWhitespace();
                continue;
            }

            if (Character.isDigit(currentChar)) {
                tokens.add(new Token(TokenType.NUMBER, number()));
                continue;
            }

            if (currentChar == ’+’) {
                tokens.add(new Token(TokenType.PLUS, "+"));
                advance();
                continue;
            }

            if (currentChar == '-') {
                tokens.add(new Token(TokenType.MINUS, "-"));
                advance();
                continue;
            }

            if (currentChar == '*') {
                tokens.add(new Token(TokenType.MULTIPLY, "*"));
                advance();
                continue;
            }

            if (currentChar == '/') {
                tokens.add(new Token(TokenType.DIVIDE, "/"));
                advance();
                continue;
            }

            if (currentChar == '(') {
                tokens.add(new Token(TokenType.LPAREN, "("));
                advance();
                continue;
            }

            if (currentChar == ')') {
                tokens.add(new Token(TokenType.RPAREN, ")"));
                advance();
                continue;
            }

            throw new RuntimeException("Unexpected character: " + currentChar);
        }

        tokens.add(new Token(TokenType.EOF, ""));
        return tokens;
    }
}
```

Explanation:

The Lexer class is responsible for breaking the input string into a list of tokens.
Advances to the next character in the input.
Skips over any whitespace characters.
Reads a sequence of digits to form a number token.
Tokenizes the entire input string into a list of tokens.
Iterates through each character of the input to classify it into tokens.

Class: Parser

```

class Parser {
    private final List<Token> tokens;
    private int pos = 0;
```

```java
    private Token currentToken;

    Parser(List<Token> tokens) {
        this.tokens = tokens;
        this.currentToken = tokens.get(pos);
    }

    private void eat(TokenType type) {
        if (currentToken.type == type) {
            pos++;
            currentToken = tokens.get(pos);
        } else {
            throw new RuntimeException("Unexpected token: " + currentToken);
        }
    }

    private Node factor() {
        Token token = currentToken;
        if (token.type == TokenType.NUMBER) {
            eat(TokenType.NUMBER);
            return new NumberNode(Integer.parseInt(token.value));
        } else if (token.type == TokenType.LPAREN) {
            eat(TokenType.LPAREN);
            Node node = expression();
            eat(TokenType.RPAREN);
            return node;
        }
        throw new RuntimeException("Unexpected token: " + token);
    }

    private Node term() {
        Node node = factor();
        while (currentToken.type == TokenType.MULTIPLY || currentToken.type ==
           TokenType.DIVIDE) {
            Token token = currentToken;
            if (token.type == TokenType.MULTIPLY) {
                eat(TokenType.MULTIPLY);
            } else if (token.type == TokenType.DIVIDE) {
                eat(TokenType.DIVIDE);
            }
            node = new BinOpNode(node, token, factor());
        }
        return node;
    }

    private Node expression() {
        Node node = term();
        while (currentToken.type == TokenType.PLUS || currentToken.type ==
           TokenType.MINUS) {
            Token token = currentToken;
            if (token.type == TokenType.PLUS) {
                eat(TokenType.PLUS);
            } else if (token.type == TokenType.MINUS) {
                eat(TokenType.MINUS);
            }
            node = new BinOpNode(node, token, term());
        }
        return node;
    }

    Node parse() {
        return expression();
    }
}
```

Explanation:

The Parser class is responsible for parsing the list of tokens and generating the AST.
Initializes the parser with the list of tokens and sets the initial token position.
the "eat" function Consumes the current token if it matches the expected type.
Parses a factor according to the grammar.
Parses and returns an expression within parentheses.
Parses a term according to the grammar.
Parses and returns a binary operation node for multiplication or division.
Parses an expression according to the grammar.
Parses and returns a binary operation node for addition or subtraction.
Parses the entire list of tokens and returns the root of the AST.

Abstract Class:Node And Class: NumberNode

```java

abstract class Node {}

class NumberNode extends Node {
    int value;

    NumberNode(int value) {
        this.value = value;
    }
}
```

Explanation:

The Node class serves as the base class for all AST nodes.
A subclass of Node representing number literals in the AST.
Initializes the node with a numeric value.

Class: BinOpNode

```java

class BinOpNode extends Node {
    Node left;
    Token op;
    Node right;

    BinOpNode(Node left, Token op, Node right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }
}
```

Explanation:

A subclass of Node representing binary operations in the AST. Initializes the node with left and right child nodes and an operator token.

Class: Token

```java



class Token {
    TokenType type;
    String value;

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

    @Override
    public String toString() {
        return "Token{" + "type=" + type + ", value='" + value + '\'' + '}';
    }
}
```

Explanation:

The Token class represents a token in the input string. Returns a string representation of the token.

Enum: TokenType

```
enum TokenType {
    NUMBER, PLUS, MINUS, MULTIPLY, DIVIDE, LPAREN, RPAREN, EOF
}
```

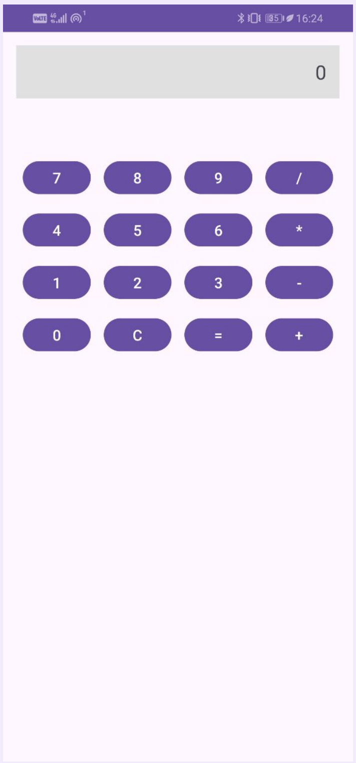
Explanation:

Defines the types of tokens that the lexer can generate.

Designing The UI With AndroidStudio

I designed the UI with AndroidStudio and I uploaded the zip file. The app looks like this on an Android phone.

Explanation:


16:24
0
7
8
9
/
4
5
6
*
1
2
3
-
0
C
=
+