

INTERPRETER CODING PROJECT-1

Name: Richa

Enrollment Number: 20114081

Group ID: 6

Project Number: 34

Problem Statement

Building an Interpreter in Java language that does subtraction. Work with tokens, lexical analyzer, and expressions that subtract integers from integers

Solution

I implemented two interpreters which will perform lexical analysis, parsing, and error handling.

The first one is the interpreter in the file **JavaSubtractionInterpreter.java**

To run this file, run:

```
javac JavaSubtractionInterpreter.java  
java JavaSubtractionInterpreter
```

Structure of the file

The interpreter works as follows:

Token class:

The class Token that represents a token, which is the smallest unit of a programming language's syntax. A token is a combination of a TokenType and a value, which represents the actual string of characters in the code that the token corresponds to.

The TokenType enum defines a set of possible types of tokens, such as PUBLIC, STATIC, CLASS, INT, VOID, MAIN, EQUALS, STRING, LBRACKET, RBRACKET, IDENTIFIER, LBRACE, INTEGER, RBRACE, LPAREN, RPAREN, SEMICOLON, EOF, PLUS, MINUS, ASTERISK, and SLASH. These tokens represent keywords, operators, symbols, and identifiers in the programming language.

The Token class has two instance variables, type and value, which represent the type and value of the token. The constructor initializes these instance variables with the values passed as arguments.

The class also provides accessor methods getType() and getValue() to retrieve the type and value of a token. The toString() method returns a string representation of a token in the format "Token(TokenType, value)". This method is useful for debugging and printing tokens to the console.

Lexer class:

This is the first step in interpreting the input program, the lexer is responsible for generating the smallest units which can be analyzed for the working of the source code.

The lexer keeps track of the next characters in the input as depending on whether the pattern of the input matches a keyword, an identifier or operators etc classifies and stores them accordingly, it also provides methods to get the nextToken, skip spaces etc.

The class Lexer is responsible for tokenizing a given input string. The Lexer has a private field 'input' that stores the input string, and fields 'position' and 'currentChar' that keep track of the current position in the input string and the current character being processed, respectively. The Lexer class provides several methods to perform the tokenization, including:

- Lexer(String input): A constructor that initializes the input, position, and currentChar fields.

-
- `integer()`: A private method that scans the input string for a sequence of digits and returns the corresponding integer value. This method advances the position field as it reads digits from the input string.
 - `advance()`: A private method that advances the position field and updates the `currentChar` field accordingly. This method is called by other methods to move to the next character in the input string.
 - `skipWhiteSpace()`: A private method that skips over whitespace characters (spaces, tabs, newlines) in the input string.
 - `identifier()`: A private method that scans the input string for a sequence of letters or digits that form an identifier (e.g., a variable name or function name). This method advances the position field as it reads characters from the input string.
 - `getNextToken()`: A public method that returns the next token in the input string. This method uses a while loop to iterate over the input string and determine the type of each token. It recognizes a set of predefined tokens, including braces, brackets, parentheses, operators, and keywords such as `public`, `int`, `static`, `void`, `main`, `String`, and `class`. For each recognized token, it advances the position field and returns a corresponding `Token` object that includes the token type and the token value. If an invalid character is encountered, it throws a `RuntimeException`. When the end of the input string is reached, it returns a `Token` object of type `EOF`.

IntegerNode, BinaryOperationNode, IdentifierNode, classDeclarationNode class:

These are the helper classes of the parser, which help in forming the abstract syntax tree structure. The ASTs are used for analyzing and interpreting code written in the language. The code also provides implementations of methods for visiting and evaluating the nodes of the AST.

The `Node` class is an abstract class that defines two abstract methods `visit()` and `evaluate()`. The `visit()` method returns a string representation of the node and its children, while the `evaluate()` method computes the value of the node.

The `IdentifierNode` class extends `Node` and represents an identifier in the programming language. It has a constructor that takes a `Token` object (which represents a lexical token in the language) and sets its value field to the value of the token. It also overrides the `visit()` method to return the value field as a string.

The `classDeclarationNode` class also extends `Node` and represents a class declaration in the programming language. It has a constructor that takes an `IdentifierNode` object for the class name and a list of `Node` objects for the body of the class. It also overrides the `visit()` method to construct and return a string representation of the class and its body.

The `IntegerNode` class extends the `Node` class and represents an integer value in the AST. It contains a `Token` object that represents the token in the input code that corresponds to the integer value, as well as the integer value itself. It provides methods to get the integer value, evaluate the node (which returns the integer value), and visit the node (which returns a string representation of the integer value).

The `BinaryOperationNode` class also extends the `Node` class and represents a binary operation (addition or subtraction) in the AST. It contains three `Node` objects that represent the left operand, the operator, and the right operand, respectively. It provides methods to evaluate the node (which returns the result of the operation) and to visit the node (which returns an empty string).

The `classDeclarationNode` class extends the `Node` class and represents a class declaration in the AST. It contains an `IdentifierNode` object that represents the class name, as well as a list of `Node` objects that represent the body of the class declaration. It provides methods to get the class name and the body, as well as to visit the node (which returns a string representation of the class declaration).

The `expr()` method parses an arithmetic expression and constructs a corresponding AST node. The `expression()` method evaluates an expression and prints the result. The `mainFunction()` method parses a main function declaration and constructs a corresponding AST node. The `classDeclaration()` method parses a class declaration and constructs a corresponding AST node. The `parse()` method is the entry point of the parser and parses the entire input code.

Interpreter class

This class contains the code for the parser. It has a constructor that takes a `Lexer` object as input, which is used to tokenize the input program. The `currentToken` variable holds the current token being processed.

The eat() method is a helper method that checks if the current token is of the expected type and moves to the next token if it is. If the current token is not of the expected type, it throws a runtime exception with an error message.

The identifier() method returns an IdentifierNode object, which represents an identifier in the program. It creates a new IdentifierNode object with the current token and then calls eat() to move to the next token.

The term() method is a recursive method that handles the terms in the program. A term is either an integer or an expression enclosed in parentheses. If the current token is an integer, it creates a new IntegerNode object with the current token and returns it. If the current token is a left parenthesis, it calls the expr() method to get the expression inside the parentheses and then calls eat() to move to the right parenthesis. If the current token is not an integer or a left parenthesis, it throws a runtime exception with an error message.

Flow of the code

The main function in the JavaSubtractionInterpreter class is responsible for starting the interpreter. It takes the input program, and then creates the lexer instance for the given input which will be responsible for tokenizing the input string. Then it calls the parse method of the interpreter which checks the basic structure of the program: consisting of a class with a main method and an arithmetic subtraction expression which needs to be evaluated. At the point when it reaches the expression, it creates the binaryOperationNode which contains the left operand, operator and the operator and when the tree is visited after creation it gets evaluated and the output is displayed to the user.

Code

[Github Link](#)

```
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
// Define the Token class
class Token {
    public enum TokenType {
        PUBLIC,
        STATIC,
        CLASS,
        INT,
        VOID,
        MAIN,
        EQUALS,
        STRING,
        LBRACKET,
        RBRACKET,
        IDENTIFIER,
        LBRACE,
        INTEGER,
        RBRACE,
        LPAREN,
        RPAREN,
        SEMICOLON,
        EOF,
        PLUS,
        MINUS,
        ASTERISK,
        SLASH
    }

    private TokenType type;
    private String value;

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

    public TokenType getType() {
        return type;
    }

    public String getValue() {
        return value;
    }

    @Override
    public String toString() {
        return String.format("Token(%s, %s)", type, value);
    }
}
```

```
// Define the Lexer class
class Lexer {
    private String input;
    private int position;
    private char currentChar;

    public Lexer(String input) {
        this.input = input;
        this.position = 0;
        this.currentChar = this.input.charAt(this.position);
    }

    private int integer() {
        StringBuilder result = new StringBuilder();
        while (this.currentChar != '\0' && Character.isDigit(this.currentChar)) {
            result.append(this.currentChar);
            this.advance();
        }
        // this.position++;
        return Integer.parseInt(result.toString());
    }

    private void advance() {
        this.position++;
        if (this.position > this.input.length() - 1) {
            this.currentChar = '\0'; // EOF
        } else {
            this.currentChar = this.input.charAt(this.position);
        }
    }

    private void skipWhiteSpace() {
        while (this.currentChar != '\0' && Character.isWhitespace(this.currentChar)) {
            this.advance();
        }
    }

    private String identifier() {
        StringBuilder result = new StringBuilder();
        while (this.currentChar != '\0' && Character.isLetterOrDigit(this.currentChar)) {
            result.append(this.currentChar);
            this.advance();
        }
        return result.toString();
    }

    public Token getNextToken() {
```

```
while (this.currentChar != '\0') {
    if (Character.isWhitespace(this.currentChar)) {
        this.skipWhiteSpace();
        continue;
    }

    if (this.currentChar == '{') {
        this.advance();
        return new Token(Token.TokenType.LBRACE, "{");
    }
    if (this.currentChar == '[') {
        this.advance();
        return new Token(Token.TokenType.LBRACKET, "[");
    }

    if (this.currentChar == '(') {
        this.advance();
        return new Token(Token.TokenType.LPAREN, "(");
    }

    if (this.currentChar == '}') {
        this.advance();
        return new Token(Token.TokenType.RBRACE, "}");
    }
    if (this.currentChar == ']') {
        this.advance();
        return new Token(Token.TokenType.RBRACKET, "]");
    }
    if (this.currentChar == ')') {
        this.advance();
        return new Token(Token.TokenType.RPAREN, ")");
    }
    if (this.currentChar == '=') {
        this.advance();
        return new Token(Token.TokenType.EQUALS, "=");
    }

    if (this.currentChar == '-') {
        this.advance();
        return new Token(Token.TokenType.MINUS, "-");
    }
    if (this.currentChar == ';') {
        this.advance();
        return new Token(Token.TokenType.SEMICOLON, ";");
    }
    if (Character.isDigit(this.currentChar)) {
        return new Token(Token.TokenType.INTEGER,
            Integer.toString(this.integer()));
    }
}
```

```
}

if (this.currentChar == 'p') {
    String keyword = this.input.substring(this.position, this.position + 6);
    if (keyword.equals("public")) {
        this.advance();
        this.advance();
        this.advance();
        this.advance();
        this.advance();
        this.advance();
        return new Token(Token.TokenType.PUBLIC, "public");
    }
}

if(this.currentChar == 'i') {
    String keyword = this.input.substring(this.position, this.position + 3);
    if (keyword.equals("int")) {
        this.advance();
        this.advance();
        this.advance();
        return new Token(Token.TokenType.INT, "int");
    }
}

if (this.currentChar == 's') {
    String keyword = this.input.substring(this.position, this.position + 6);
    if (keyword.equals("static")) {
        this.advance();
        this.advance();
        this.advance();
        this.advance();
        this.advance();
        this.advance();
        return new Token(Token.TokenType.STATIC, "static");
    }
}

if (this.currentChar == 'v') {
    String keyword = this.input.substring(this.position, this.position + 4);
    if (keyword.equals("void")) {
        this.advance();
        this.advance();
        this.advance();
        this.advance();
        return new Token(Token.TokenType.VOID, "void");
    }
}
```

```
    if (this.currentChar == 'm') {
        String keyword = this.input.substring(this.position, this.position + 4);
        if (keyword.equals("main")) {
            this.advance();
            this.advance();
            this.advance();
            this.advance();
            return new Token(Token.TokenType.MAIN, "main");
        }
    }

    if (this.currentChar == 'S') {
        String keyword = this.input.substring(this.position, this.position + 6);
        if (keyword.equals("String")) {
            this.advance();
            this.advance();
            this.advance();
            this.advance();
            this.advance();
            this.advance();
            return new Token(Token.TokenType.STRING, "String");
        }
    }

    if (this.currentChar == 'c') {
        String keyword = this.input.substring(this.position, this.position + 5);
        // System.out.println("key:" + keyword + " " + this.position);
        if (keyword.equals("class")) {
            this.advance();
            this.advance();
            this.advance();
            this.advance();
            this.advance();
            return new Token(Token.TokenType.CLASS, "class");
        }
    }

    if (Character.isLetter(this.currentChar)) {
        return new Token(Token.TokenType.IDENTIFIER, this.identifier());
    }

    throw new RuntimeException("Invalid character");
}

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

```
// Define the Node classes
abstract class Node {
    public abstract String visit();
    public abstract int evaluate();
}

class IdentifierNode extends Node {
    private Token token;
    private String value;

    public IdentifierNode(Token token) {
        this.token = token;
        this.value = this.token.getValue();
    }

    public String getValue() {
        return value;
    }

    @Override
    public String visit() {
        return this.value;
    }
    @Override
    public int evaluate() {
        return 0;
    }
}

}

class classDeclarationNode extends Node {
    private IdentifierNode className;
    // private Token classKeyword;
    private List<Node> body;

    public classDeclarationNode(IdentifierNode className, List<Node> body) {
        // this.classKeyword = classKeyword;
        this.className = className;
        this.body = body;
    }

    public Node getClassName() {
        return className;
    }

    public List<Node> getBody() {
```

```

        return body;
    }

    public String toString() {
        return String.format("classDeclarationNode( %s, %s)", className.getValue(),
body);
    }

    @Override
    public int evaluate() {
        return 0;
    }
    @Override
    public String visit() {
        StringBuilder sb = new StringBuilder();
        sb.append("class ");
        sb.append(this.className.visit());
        sb.append(" {\n");
        for (Node node : this.body) {
            sb.append(node.visit());
        }
        sb.append("}\n");
        return sb.toString();
    }
}

// Define the Interpreter class
class Interpreter {
    private Lexer lexer;
    private Token currentToken;

    public Interpreter(Lexer lexer) {
        this.lexer = lexer;
        this.currentToken = this.lexer.getNextToken();
    }

    private void eat(Token.TokenType tokenType) {
        // System.out.println(this.currentToken.getValue());
        if (this.currentToken.getType() == tokenType) {
            this.currentToken = this.lexer.getNextToken();
        } else {
            throw new RuntimeException("Invalid syntax");
        }
    }

    private IdentifierNode identifier() {
        IdentifierNode node = new IdentifierNode(this.currentToken);

```

```

        this.eat(Token.TokenType.IDENTIFIER);
        return node;
    }

    private Node term() {
        Token token = this.currentToken;
        if (token.getType() == Token.TokenType.INTEGER) {
            this.eat(Token.TokenType.INTEGER);
            return new IntegerNode(token);
        } else if (token.getType() == Token.TokenType.LPAREN) {
            this.eat(Token.TokenType.LPAREN);
            Node node = this.expr();
            this.eat(Token.TokenType.RPAREN);
            return node;
        } else {
            throw new RuntimeException("Invalid syntax");
        }
    }
}

class IntegerNode extends Node {
    private Token token;
    private int value;

    public IntegerNode(Token token) {
        this.token = token;
        this.value = Integer.parseInt(this.token.getValue());
    }

    public int getValue() {
        return value;
    }

    @Override
    public int evaluate() {
        return this.value;
    }

    @Override
    public String visit() {
        return Integer.toString(this.value);
    }
}

public Node expr() {
    Node node = this.term();
    // System.out.println(this.currentToken.getType());
    while (this.currentToken.getType() == Token.TokenType.PLUS ||
        this.currentToken.getType() == Token.TokenType.MINUS) {

```

```

        Token token = this.currentToken;
        if (token.getType() == Token.TokenType.PLUS) {
            this.eat(Token.TokenType.PLUS);
        } else if (token.getType() == Token.TokenType.MINUS) {
            this.eat(Token.TokenType.MINUS);
        }

        node = new BinaryOperationNode(node, token, this.term());
    }

    return node;
}

```

```

private Node expression() {
    Node node = this.expr();
    System.out.println("Result: " + node.evaluate());
    return node;
}

```

```

private Node mainFunction() {
    this.eat(Token.TokenType.PUBLIC);
    this.eat(Token.TokenType.STATIC);
    this.eat(Token.TokenType.VOID);
    this.eat(Token.TokenType.MAIN);
    this.eat(Token.TokenType.LPAREN);
    this.eat(Token.TokenType.STRING);
    this.eat(Token.TokenType.LBRACKET);
    this.eat(Token.TokenType.RBRACKET);
    this.eat(Token.TokenType.IDENTIFIER);
    this.eat(Token.TokenType.RPAREN);
    this.eat(Token.TokenType.LBRACE);
    this.eat(Token.TokenType.INT);
    this.eat(Token.TokenType.IDENTIFIER);
    this.eat(Token.TokenType.EQUALS);
    Node node = this.expression();
    this.eat(Token.TokenType.SEMICOLON);
    this.eat(Token.TokenType.RBRACE);
    return node;
}

```

```

private Node classDeclaration() {
    this.eat(Token.TokenType.PUBLIC);
    this.eat(Token.TokenType.CLASS);
    IdentifierNode classNameNode = this.identifier();
    List<Node> body = new ArrayList<>();
    this.eat(Token.TokenType.LBRACE);
}

```

```

        body.add(this.mainFunction());
        this.eat(Token.TokenType.RBRACE);
        return new classDeclarationNode(classNameNode, body);
    }

    public void parse() {
        Node node = this.classDeclaration();
    }
}

// Define the BinOpNode class
class BinaryOperationNode extends Node {
    private Node left;
    private Token operator;
    private Node right;

    public BinaryOperationNode(Node left, Token operator, Node right) {
        this.left = left;
        this.operator = operator;
        this.right = right;
    }

    public int evaluate() {
        int leftVal = Integer.parseInt(this.left.visit());
        int rightVal = Integer.parseInt(this.right.visit());

        switch (this.operator.getType()) {
            case PLUS:
                return leftVal + rightVal;
            case MINUS:
                return leftVal - rightVal;
            default:
                throw new RuntimeException("Invalid operator");
        }
    }

    @Override
    public String visit() {
        return "";
    }
}

// Example usage
public class JavaSubtractionInterpreter {
    public static void main(String[] args) {

```

//change the input expression here: note for handling complex expressions like different variables names and then operating we can modify the way expr() is handled

```
String input = "public class MyClass {\n" +  
    "    public static void main(String[] args) {\n" +  
    "        int x = 10 - 6;\n" +  
    "    }\n" +  
    "}\n";  
  
Lexer lexer = new Lexer(input);  
Interpreter interpreter = new Interpreter(lexer);  
interpreter.parse();  
}  
}
```

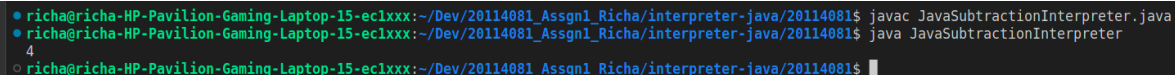
TestCases

Change the expression in the line

```
"    int x = 10 - 6;\n" +
```

of the input to perform subtraction

Output



```
richa@richa-HP-Pavilion-Gaming-Laptop-15-eclxxx:~/Dev/20114081_Assgn1_Richa/interpreter-java/20114081$ javac JavaSubtractionInterpreter.java  
richa@richa-HP-Pavilion-Gaming-Laptop-15-eclxxx:~/Dev/20114081_Assgn1_Richa/interpreter-java/20114081$ java JavaSubtractionInterpreter  
4  
richa@richa-HP-Pavilion-Gaming-Laptop-15-eclxxx:~/Dev/20114081_Assgn1_Richa/interpreter-java/20114081$
```

The second interpreter implemented is for a dummy programming language and is fully functional with 'functions', 'loops', 'arithmetic operators', 'logical and relational operators', 'print' statements, performs any arithmetic operation be it with decimals, negative numbers, positive numbers etc.

Working:

Link:

<https://github.com/Richa-iitr/Compiler-Design-Project-Interpreter/tree/main/dummy-language-interpreter-in-java>

Folder: **dummy-language-interpreter-in-java**

The code contains different modules dedicated for lexer, parser and ast tree. It contains the regex of the valid tokens, keywords, identifiers in the file:

<src/main/resources/lexical-spec.lex>

```
//type      regex

//keywords
IF      "if"    0
ELSE    "else"  0
FOR      "for"   0
WHILE    "while" 0
PRINT    "print" 0
PRINTLN  "println" 0
RETURN   "return" 0
CONTINUE "continue" 0
BREAK    "break" 0

//type keywords
INT      "int"    0
FLOAT    "float"  0
BOOLEAN  "boolean" 0
STRING   "string" 0
VOID     "void"   0

//constants
INT_CONST "[0-9]+"      1
FLOAT_CONST "([0-9]+([.][0-9]*)?|[.][0-9]+)" 1
BOOL_CONST  "true|false" 1
STRING_CONST "\\\"(\\\\\\\\.|[^\"])*\\\"" 1
//identifier
IDENTIFIER  "[_a-zA-Z][_a-zA-Z0-9]*" 1

//symbols
(      "\\("  0
)      "\\)"  0
{      "\\{"  0
}      "\\}"  0
'      "\\'"  0
;      "\\;"  0
[      "\\["  0
]      "\\]"  0

//assignment operators
=      "=="  0
+=      "\\+="  0
-=      "\\-=" 0
*=      "\\*=" 0
/=      "\\!=" 0
%=      "\\%=" 0
```

```

+      "\\+"  0

//operators
-      "-"    0
*      "\\*"  0
/      "/"    0
%      "%"    0
!      "!"    0
++     "\\++"  0
--     "--"   0
EQ_OP  "=="   0
NEQ_OP "!="   0
GE_OP  ">="   0
LE_OP  "<="   0
AND_OP  "&&"   0
OR_OP  "\\||"  0
>      ">"    0
<      "<"    0

```

Like in the previous code it does lexical analysis, then parsing using ast, errors are handled separately using the dedicated exception files in each folder like the [src/main/java/com/richa/interpreter/lex/LexicalException.java](#)

The visitor files visit the node of the Abstract Syntax Tree and the Main.java file is responsible for actually running the code.

The grammar of the language is similar to C except it doesn't contain semicolon and instead of print function uses print and println as keywords, here are the testcases which can be found in the file under [src/main/resources/test.c+](#)- The testcases display all the grammar of the language.

```
int[] main() {
```

```

    int x = -15
    int y = 4

    println "Subtract -15 - 4"
    int res = x - y
    println "result:" + res

    println ""

```

```
    int fib = 20
```

```
    println "fibonacci:"
```

Change this to test
different expressions

```
    print "fibdp(" + fib + ") = "  
    println fibdp(fib)  
  
    print "fib(" + fib + ") = "  
    println fib(fib)  
  
    println ""  
    println "exponentiation by squaring:"  
  
    int base = 3  
    int exp = 0  
  
    print "exp(" + base + ", " + exp + ") = "  
    println exp(base, exp)  
  
    println ""  
  
    int n = 10  
  
    println "factorial:"  
    print "fact(" + n + ") = "  
    println fact(n)  
  
    println ""  
  
    //other tests for break, continue and type casting  
    int k  
    for(k = 0; k < 20; k++) {  
        if(k == 5) break  
        print k + "\n"  
    }  
  
    println ""  
  
    println (int) 3.54  
    println (float) 3 / 2  
  
    int arr[20]  
    for(k = 0; k < len(arr); k++) {  
        arr[k] = k + 1  
    }  
  
    println "arr length " + len(arr)  
  
    return arr  
}
```

```

//recursive fibonacci, because now we can!
int fib(int n) {
    if(n == 0 || n == 1)
        return n

    return fib(n - 1) + fib(n - 2)
}

//fibonacci with no recursion
int fibdp(int n) {
    int fib[3]
    fib[0] = 1 fib[1] = 1

    int i
    for(i = 2; i < n; i++)
        fib[i % 3] = fib[(i - 1) % 3] + fib[(i - 2) % 3]

    return fib[(i - 1) % 3]
}

//exponentiation by squaring
float exp(float base, float exp) {
    float y = 1
    if(exp < 0) {
        base = 1 / base
        exp = -exp
    }

    if(exp != 0) {
        while(exp > 1) {
            if(exp % 2 == 0) {
                base *= base
                exp /= 2
            } else {
                y *= base
                base *= base
                exp = (exp - 1) / 2
            }
        }
        return base * y
    } else
        return 1
}

int fact(int n) {
    if(n == 0) return 1
    return n * fact(n - 1)
}

```

Output

```
richa@richa-HP-Pavilion-Gaming-Laptop-15-eclxxx:~/Dev/20114081_Assgn1_Richa/interpreter-java/20114081/dummy-language-interpreter-in-java$ java -jar target/interpreter-1.0.jar src/main/resources/test.c+-
Subtract -15 - 4
result:-19

fibonacci:
fibdp(20) = 6765
fib(20) = 6765

exponentiation by squaring:
exp(3, 0) = 1.0

factorial:
fact(10) = 3628800

0
1
2
3
4

3
1.5
arr length 20
```

Run

Compile it using the maven command: (If maven not installed install using `sudo apt install maven` in ubuntu)

```
mvn package
```

A jar file will be created in the target folder. Then, to run the interpreter, type in the terminal:

```
java -jar target/interpreter-1.0.jar src/main/resources/test.c+-
```