Лабораторная работа № 1.2. «Лексический анализатор на основе

регулярных выражений» % 14 марта 2023 г. % Самохвалова Полина, ИУ9-62Б

Цель работы

Целью данной работы является приобретение навыка разработки простейших лексических анализаторов, работающих на основе поиска в тексте по образцу, заданному регулярным выражением.

Индивидуальный вариант

Химические вещества: последовательности латинских букв и цифр, начинающиеся с заглавной буквы, при этом после цифры не может следовать строчная буква (атрибут: строка). Примеры: «CuSO4», «CH3CH2OH», «Fe2O3».

Коэффициенты: последовательности десятичных цифр. Между коэффициентом и веществом пробел может отсутствовать. Операторы: «+», «->».

Реализация

Main.java

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public class Main {
    public static void main(String[] args) throws IOException {
        String text = new
                String(Files.readAllBytes(Paths.get(args[0])));
        Scanner scanner = new Scanner(text);
        Token token = scanner.nextToken();
        while (token.tag != DomainTag.END_OF_PROGRAM) {
            System.out.println(token);
```

```
            token = scanner.nextToken();
        }
    }
}
```

DomainTag.java

```
enum DomainTag {
    SUBSTANCE, NUMBER, OPERATOR, UNKNOWN, END_OF_PROGRAM
}
```

Token.java

```
abstract public class Token {
    public final DomainTag tag;
    public final Position starting;

    protected Token(DomainTag tag, Position starting) {
        this.tag = tag;
        this.starting = starting;
    }

    @Override
    public String toString() {
        return starting.toString();
    }
}
```

SubstanceToken.java

```
public class SubstanceToken extends Token {
    public final String value;

    protected SubstanceToken(String value, Position starting) {
        super(DomainTag.SUBSTANCE, starting);
        this.value = value;
    }

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

NumberToken.java

```
public class NumberToken extends Token {
    public final String value;
```

```
    protected NumberToken(String value, Position starting) {
        super(DomainTag.NUMBER, starting);
        this.value = value;
    }

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

OperatorToken.java

```
public class OperatorToken extends Token {
    public final String value;

    protected OperatorToken(String value, Position starting) {
        super(DomainTag.OPERATOR, starting);
        this.value = value;
    }

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

UnknownToken.java

```
public class UnknownToken extends Token {

    public UnknownToken(DomainTag tag, Position starting) {
        super(tag, starting);
    }

    @Override
    public String toString() {
        return "syntax error " + super.toString();
    }
}
```

Scanner.java

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

public class Scanner {
    private Position cur;
```

```
public Scanner(String text) {
    cur = new Position(text);
}

public Token nextToken() {

    String substance =
            "[A-Z](([0-9]+[A-Z]+)|([A-Z]+)|([a-z]+))*[0-9]*";
    String number = "[0-9]+";
    String operator = "(\\+)|(->)";
    String pattern = "(?<substance>^" + substance + ")|(?<number>^"
            + number + ")|(?<operator>^" + operator + ")";

    Pattern p = Pattern.compile(pattern);
    Matcher m;

    while (!cur.isEndOfFile()) {
        m = p.matcher(cur.getText());
        if (cur.isWhitespace()) {
            cur = cur.next();
        } else if (m.find()) {
            String m_substance = m.group("substance");
            String m_number = m.group("number");
            String m_operator = m.group("operator");
            Token token = null;
            if (m_substance != null) {
                token = new SubstanceToken(m_substance, cur);
                cur = cur.next_x(m_substance.length());
            } else if (m_number != null) {
                token = new NumberToken(m_number, cur);
                cur = cur.next_x(m_number.length());
            } else if (m_operator != null) {
                token = new OperatorToken(m_operator, cur);
                cur = cur.next_x(m_operator.length());
            }
            return token;
        } else {
            Token token = new UnknownToken(DomainTag.UNKNOWN, cur);
            while (!cur.isEndOfFile() &&
                    !p.matcher(cur.getText()).find() &&
                    !cur.isWhitespace()) {
                cur = cur.next();
            }
            return token;
        }
    }
```

```
            return new UnknownToken(DomainTag.END_OF_PROGRAM, cur);

        }
    }
```

Position.java

```
public class Position {

    private String text;
    private int line, pos;
    public String getText() {
        return text;
    }
    public int getLine() {
        return line;
    }
    public int getPos() {
        return pos;
    }

    public Position(String text) {
        this.text = text;
        line = pos = 1;
    }

    public Position(Position p) {
        this.text = p.getText();
        this.line = p.getLine();
        this.pos = p.getPos();
    }

    @Override
    public String toString() {
        return "(" + line + ", " + pos + ')';
    }

    public boolean isEndOfFile() {
        return text.length() == 0;
    }

    public int getCode() {
        return isEndOfFile() ? -1 : text.codePointAt(0);
    }

    public boolean isNewLine() {
        if (isEndOfFile()) {
```

```
            return true;
        } else {
            return (text.charAt(0) == '\n');
        }
    }

    public boolean isWhitespace() {
        return !isEndOfFile() && Character.isWhitespace(getCode());
    }
    public Position next() {
        Position p = new Position(this);
        if (!p.isEndOfFile()) {
            if (p.isNewLine()) {
                p.line++;
                p.pos = 1;
            } else {
                if (Character.isHighSurrogate(p.text.charAt(0))) {
                    p.text = p.text.substring(1);
                }
                p.pos++;
            }
            p.text = p.text.substring(1);
        }
        return p;
    }

    public Position next_x(int x) {
        Position p = new Position(this);
        for (int i = 0; i < x; i++) {
            p = p.next();
        }
        return p;
    }

}
```

Тестирование

Входные данные:

```
CaCO3 -> CaO + CO2
      2Mg + O2 -> 2MgO
   Fe2O3 + 2Al -> Al2O3 + 2Fe
NaOH + HCl -> NaCl + H2O
```

Вывод на `stdout`:

```
SUBSTANCE (1, 1): CaCO3
OPERATOR (1, 7): ->
SUBSTANCE (1, 10): CaO
OPERATOR (1, 14): +
SUBSTANCE (1, 16): CO2
NUMBER (2, 7): 2
SUBSTANCE (2, 8): Mg
OPERATOR (2, 11): +
SUBSTANCE (2, 13): O2
OPERATOR (2, 16): ->
NUMBER (2, 19): 2
SUBSTANCE (2, 20): MgO
SUBSTANCE (3, 4): Fe2O3
OPERATOR (3, 10): +
NUMBER (3, 12): 2
SUBSTANCE (3, 13): Al
OPERATOR (3, 16): ->
SUBSTANCE (3, 19): Al2O3
OPERATOR (3, 25): +
NUMBER (3, 27): 2
SUBSTANCE (3, 28): Fe
SUBSTANCE (4, 1): NaOH
OPERATOR (4, 6): +
SUBSTANCE (4, 8): HCl
OPERATOR (4, 12): ->
SUBSTANCE (4, 15): NaCl
OPERATOR (4, 20): +
SUBSTANCE (4, 22): H2O
```

Вывод

В результате выполнения лабораторной работы был приобретен навык разработки простейших лексических анализаторов, работающих на основе поиска в тексте по образцу, заданному регулярным выражением. На языке Java были реализованы две первые фазы стадии анализа: чтение входного потока и лексический анализ. Чтение входного потока осуществляется из файла, при этом лексический анализатор вычисляет текущие координаты в обрабатываемом тексте. В результате работы программы в стандартный поток вывода выдаются описания распознанных лексем.