Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ имени Н.Э.БАУМАНА
(национальный исследовательский университет)»

Факультет: Информатика и системы управления

Кафедра: Теоретическая информатика и компьютерные технологии

Лабораторная работа № 3

Раскрутка самоприменимого компилятора

Вариант 2

Работу выполнил

студент группы ИУ9-61

Бакланова А.Д.

Москва, 2020

Цель работы:

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

Исходные данные:

Стандартная библиотека любого современного языка программирования содержит средства для поиска в тексте образцов, заданных регулярными выражениями. При этом используется расширенный синтаксис записи регулярных выражений, позволяющий по сути выйти за рамки регулярных языков. Механизм поиска по таким регулярным выражениям годится для написания простейших лексических анализаторов. Однако, для этого механизма характерна нелинейная зависимость времени работы от длины распознаваемой лексемы, поэтому в промышленных компиляторах он не используется.

В качестве языка реализации в данной лабораторной работе выберем язык Java, стандартная библиотека которого содержит пакет `java.util.regex`, в котором располагаются классы `Pattern` и `Matcher`, предназначенные для поиска по регулярным выражениям. Документация по этому пакету находится по адресу:

`http://docs.oracle.com/javase/7/docs/api/java/util/regex/package-summary.html`.

Вводную статью по синтаксису регулярных выражений можно прочитать здесь:

`http://www.quizful.net/post/Java-RegExp`.

Идея лексического анализа на основе поиска по регулярным выражениям состоит в использовании групп, представляющих собой фрагменты регулярных выражений, заключённые в круглые скобки, значения которых запоминаются при сопоставлении текста с образцом. Например, на листинге 1 показано, как с использованием групп отличить идентификаторы от числовых литералов.

Индивидуальный вариант:

2	Комментарии: начинаются с «`/*`», заканчиваются на «`*/`» и могут пересекать границы строк текста. Идентификаторы: последовательности латинских букв и десятичных цифр, в которых буквы и цифры чередуются. Ключевые слова: «`for`», «`if`», «`m1`».

Задание:

В лабораторной работе необходимо реализовать на языке Java две первые фазы стадии анализа: чтение входного потока и лексический анализ. Чтение входного потока должно осуществляться из файла (в UTF-8), при этом лексический анализатор должен вычислять текущие координаты в обрабатываемом тексте. В результате работы программы в стандартный поток вывода должны выдаваться описания распознанных лексем в формате

```
Тег (координаты): значение
```

Например,

```
IDENT (1, 2): count
ASSIGN (1, 8): :=
NUMBER (1, 11): 100
```

Лексемы во входном файле могут разделяться пробельными символами (пробел, горизонтальная табуляция, маркеры окончания строки), а могут быть записаны слитно (если это не приводит к противоречиям).

Идентификаторы и числовые литералы не могут содержать внутри себя пробельных символов, если в задании явно не указано иного (варианты 4, 14 и 36). Комментарии, строковые и символьные литералы могут содержать внутри себя пробельные символы.

Входной файл может содержать ошибки, при обнаружении которых лексический анализатор должен выдавать сообщение с указанием координаты:

```
syntax error (10,2)
```

После обнаружения ошибки лексический анализатор должен восстанавливаться по следующей схеме: из входного потока пропускаются все подряд идущие символы до нахождения следующей лексемы.

Лексический анализатор должен иметь программный интерфейс для взаимодействия с парсером. Рекомендуется реализовывать его как итератор с методом `nextToken()` для императивных языков или функцию, возвращающую список лексем, для функциональных языков.

В регулярных выражениях рекомендуется использовать классы символов Unicode для обозначения букв, чисел и других подобных множеств. Многие движки регулярных выражений для задания классов используют синтаксис `\p{класс}`. Вместо нумерованных групп рекомендуется использовать именованные `(?<имя>regexp)`, при использовании нумерованных групп ненумеруемые обозначаются как `(?:regexp)`.

Варианты языков для лексического анализа приведены в таблицах 1, 2, 3, 4 и 5.

Реализация:

```java
import java.util.Arrays;
import java.util.Scanner;
import java.util.function.IntPredicate;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Position {
    private String text;
    private int amountRemoved;
    private int index, line, col;

    public Position(String text) {
        this(text, 0, 1, 1);
    }

    private Position(String text, int index, int line, int col) {
        this.text = text;
        this.index = index;
        this.line = line;
        this.col = col;
    }

    public boolean isEmpty() {
        return text.isEmpty();
    }

    public int getChar() {
        return index < text.length() ? text.codePointAt(index) : -1;
    }

    public boolean satisfies(IntPredicate p) {
        return p.test(getChar());
    }

    public Position skip() {
        int c = getChar();
        switch (c) {
            case -1:
                return this;
            case '\n':
                return new Position(text.substring(1), index/* + 1*/, line + 1, 1);
            default:
                return new Position(text.substring(1), index/* + (c > 0xFFFF ? 2 : 1)*/, line, col + 1);
        }
    }

    public Position skipToken(int range) {
        Position pos = this;
        for (int i = 0; i < range; i++) {
            pos = pos.skip();
        }
        pos.text = text.substring(range);

        return pos;
    }

     public Position skipWhile(IntPredicate p) {
         Position pos = this;
         while (pos.satisfies(p)) pos = pos.skip();
         return pos;
     }

     public String getText() {
         return text;
     }

     public String toString() {
         return String.format("(%d, %d)", line, col);
     }
 }
```

```
}

class SyntaxError extends Exception {
    public SyntaxError(Position pos, String msg) {
        super(String.format("Syntax error at %s: %s", pos.toString(), msg));
    }
}


enum Tag {
    IDENT,
    comments,
    sm1,
    sif,
    sfor,
    END_OF_TEXT;

    public String toString() {
        switch (this) {
            case IDENT:
                return "IDENT";
            case comments:
                return "comments";
            case sm1:
                return "sm1";
            case sif:
                return "sif";
            case sfor:
                return "sfor";
            case END_OF_TEXT:
                return "END OF TEXT";
        }
        throw new RuntimeException("unreachable code");
    }
}

class Token {
    private Tag tag;
    private String value;
    private Position start, follow;
    private Pattern pattern;

    public Token(String text, Pattern pattern) throws SyntaxError {
        this(new Position(text), pattern);
    }

    private Token(Position cur, Pattern pattern) throws SyntaxError {
        this.pattern = pattern;
        start = cur.skipWhile(Character::isWhitespace);
        follow = start.skip();

        Matcher matcher = pattern.matcher(start.getText());

        if (start.isEmpty()) {
            tag = Tag.END_OF_TEXT;
            value = "EOF";
       } else if (!matcher.find()) {
           throwError("invalid character");
       } else if (matcher.group("ident") != null) {
           tag = Tag.IDENT;
           value = matcher.group("ident");
           follow = follow.skipToken(matcher.end() - 1);
       } else if (matcher.group("sm1") != null) {
           tag = Tag.sm1;
           value = matcher.group("sm1");
           follow = follow.skipToken(matcher.end() - 1);
       } else if (matcher.group("sif") != null) {
```

```
            tag = Tag.sif;
            value = matcher.group("sif");
            follow = follow.skipToken(matcher.end() - 1);
        } else if (matcher.group("comments") != null) {
            tag = Tag.comments;
            value = matcher.group("comments");
            follow = follow.skipToken(matcher.end() - 1);
        } else if (matcher.group("sfor") != null) {
            tag = Tag.sfor;
            value = matcher.group("sfor");
            follow = follow.skipToken(matcher.end() - 1);
        } else {
            throwError("invalid character");
        }
    }

    public void throwError(String msg) throws SyntaxError {
        throw new SyntaxError(start, msg);
    }

    public boolean matches(Tag... tags) {
        return Arrays.stream(tags).anyMatch(t -> tag == t);
    }

    public Token next() throws SyntaxError {
        return new Token(follow, pattern);
    }

    public Tag getTag() {
        return tag;
    }

    public Position getPosition() {
        return start;
    }

    public String getValue() {
        return value;
    }
}

public class Lab3 {

    private static Token token;
    private static String ident = "(((^(([A-z])(\\d))+)([A-z])?)|((^((\\d)([A-z]))+)([0-9])?))";
    private static String comments = "(\\\")?(?!\\\")(.*)(\\\\)?(?=\\\")|(?!\\\")(.)";
    private static String sfor = "for";
    private static String sif = "if";
    private static String sm1 = "m1";
    private static String pattern =
            "(?<sfor>^" + sfor + ")|" +
            "(?<sif>^" + sif + ")|" +
            "(?<sm1>^" + sm1 + ")|" +
            "(?<ident>^" + ident + ")|" +
            "(?<comments>^" + comments + ")|";
  public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      in.useDelimiter("\\Z");
      String text = in.next();
      Pattern p = Pattern.compile(pattern);
      try {
          token = new Token(text, p);
          do {
              printToken(
                      token.getTag().toString(),
                      token.getPosition().toString(),
                      token.getValue()
              );
              token = token.next();

          } while (token.getTag() != Tag.END_OF_TEXT);
      } catch (SyntaxError syntaxError) {
          syntaxError.printStackTrace();
      }
  }

  private static void printToken(String tag, String position, String value) {
      System.out.printf("%s %s: %s\n", tag, position, value);
  }
}
```

Тестирование:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
/* AAAAAAA */ if for m1 a5v6n  /*b*/
^D
comments (1, 1): /
comments (1, 2): *
comments (1, 4): A
comments (1, 5): A
comments (1, 6): A
comments (1, 7): A
comments (1, 8): A
comments (1, 9): A
comments (1, 10): A
comments (1, 12): *
comments (1, 13): /
sif (1, 15): if
sfor (1, 18): for
sm1 (1, 22): m1
IDENT (1, 25): a5v6n
comments (1, 32): /
comments (1, 33): *
comments (1, 34): b
comments (1, 35): *
comments (1, 36): /

Process finished with exit code 0
```

Вывод:

В результате выполнения лабораторной работы были получены навыки разработки простейших лексических анализаторов, работающих на основе поиска в тексте по образцу, заданному регулярным выражением, также была реализована программа, считывающая строку и выводящая описания распознанных лексем.