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 № 4

Раскрутка самоприменимого компилятора

Вариант 15

Работу выполнил

студент группы ИУ9-61

Бакланова А.Д.

Москва, 2020

Цель работы:

Целью данной работы является приобретение навыка реализации лексического анализатора на объектно-ориентированном языке без применения каких-либо средств автоматизации решения задачи лексического анализа.

Индивидуальный вариант:

15	Идентификаторы: последовательности латинских букв и цифр, начинающиеся с буквы. Знаки операций: либо последовательности, состоящие из знаков `!`, `#`, `$`, `%`, `&`, `*`, `+`, `.`, `/`, `<`, `=`, `>`, `?`, `@`, `\`, `^`, `|`, `-` и `~`, либо идентификаторы, записанные в обратных кавычках (например, «`` `plus` ``»). Ключевые слова «`where`», «`->`», «`=>`».

Задание:

В лабораторной работе предлагается реализовать на языке Java две первые фазы стадии анализа: чтение входного потока и лексический анализ. При этом следует придерживаться схемы реализации объектно-ориентированного лексического анализатора, рассмотренной на лекции.

Входной поток должен загружаться из файла (в UTF-8). В результате работы программы в стандартный поток вывода должны выдаваться описания распознанных лексем в формате

```
Тег (координаты_фрагмента): атрибут лексемы
```

При этом для лексем, не имеющих атрибутов, нужно выводить только тег и координаты. Например,

```
IDENT (1, 2)-(1, 4): str
ASSIGN (1, 8)-(1, 9):
STRING (1, 11)-(1, 16): qwerty
```

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.

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

Реализация:

```java
import java.util.Arrays;
import java.util.Scanner;
import java.util.function.IntPredicate;

class Position {
    private String text;
    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 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, index + 1, line + 1, 1);
            default:
                return new Position(text, index + (c > 0xFFFF ? 2 : 1), line, col + 1);
        }
    }

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

    public String getSubText(Position end) {
        return text.substring(index, end.index);
    }

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

    public Position getCopy() {
        return new Position(text, index, line, col);
    }
```

```
59  class SyntaxError extends Exception {
60      public SyntaxError(Position pos, String msg) {
61          super(String.format("Syntax error at %s: %s", pos.toString(), msg));
62      }
63  }
64
65  enum Tag {
66      IDENT,
67      KEYWORD,
68      OPERATION,
69      UNKNOWN,
70      END_OF_TEXT;
71
72      public String toString() {
73          switch (this) {
74              case IDENT:
75                  return "IDENT";
76              case KEYWORD:
77                  return "KEYWORD";
78              case OPERATION:
79                  return "OPERATION";
80              case UNKNOWN:
81                  return "UNKNOWN";
82              case END_OF_TEXT:
83                  return "END OF TEXT";
84          }
85          throw new RuntimeException("unreachable code");
86      }
87  }
88
89  class Token {
90      private Tag tag;
91      private String value;
92      private Position start, follow;
93
94      public Token(String text) throws SyntaxError {
95          this(new Position(text));
96      }
97
98      private Token(Position cur) throws SyntaxError {
99          value = "";
100         start = cur.skipWhile(Character::isWhitespace);
101         follow = start.skip();
102         int c = start.getChar();
103         if (c == -1) {
104             tag = Tag.END_OF_TEXT;
105         }
106
107         else if (((c == '-' || c == '=' ) && follow.getChar() == '>')) {
108             follow = follow.skip();
109             assignTagKEYWORDAndValue();
110         }
111         else if (isCharacterPart(c)) {
112             follow = follow.skipWhile(isSymbol());
113             assignTagOPERATIONAndValue();
114         }
```

```
        else if (c == '`' && follow.satisfies(Character::isLetter)) {
            Position tmp = follow.skipWhile(isIdent());
            if (tmp.getChar() != '`') {
                follow = follow.skipWhile(isQuote());
                skipUnknown();
            } else {
                follow = follow.skipWhile(isQuote());
                follow = follow.skip();
                assignTagOPERATIONAndValue();
            }

        }
        else if (start.satisfies(Character::isLetter)) {
            Position p = start.getCopy();
            if (!follow.satisfies(Character::isWhitespace)) {
                if (follow.satisfies(Character::isDigit) || follow.satisfies(Character::isLetter)) {
                    follow = follow.skipWhile(Character::isLetterOrDigit);
                    String word = (p.getSubText(follow));
                    if (word.equals("where")) {
                        if (!follow.satisfies(Character::isWhitespace)) {
                            skipUnknown();
                            return;
                        }
                        assignTagKEYWORDAndValue();
                    } else
                    if (!follow.satisfies(Character::isWhitespace)) {
                        skipUnknown();
                        return;
                    }
                    else {
                        assignTagIDENTAndValue();
                    }
                } else {
                    assignTagIDENTAndValue();
                }
            } else {
                assignTagIDENTAndValue();
            }
        }

        else {
            skipUnknown();
            return;
        }
    }

    private boolean isCharacterPart(int c) {
        if (!(c == '!' || c == '#' || c == '$' || c == '%' || c == '&'
                || c == '*' || c == '+' || c == '.' || c == '/' || c == '<'
                || c == '=' || c == '>' || c == '?' || c == '@' || c == '^'
                || c == '|' || c == '-' || c == '~' || c == '\\')) {
            return false;
        }
        return true;
    }

    private IntPredicate isSymbol() {
        IntPredicate predicate = c -> c == '!' || c == '#' || c == '$' || c == '%' || c == '&'
                || c == '*' || c == '+' || c == '.' || c == '/' || c == '<'
                || c == '=' || c == '>' || c == '?' || c == '@' || c == '^'
                || c == '|' || c == '-' || c == '~' || c == '\\';
        return predicate;
    }

    private IntPredicate isQuote() {
        IntPredicate predicate = c -> (c != '`');
        return predicate;
    }

    private IntPredicate isIdent() {
        IntPredicate predicate = c -> (c != '`' && !isCharacterPart(c) && c != ' ');
        return predicate;
    }

    private void assignTagOPERATIONAndValue() {
        tag = Tag.OPERATION;
        value = start.getSubText(follow);
    }
```

```
194     private void assignTagKEYWORDAndValue() {
195         tag = Tag.KEYWORD;
196         value = start.getSubText(follow);
197     }
198 
199     private void assignTagIDENTAndValue() {
200         tag = Tag.IDENT;
201         value = start.getSubText(follow);
202     }
203 
204     private void skipUnknown() {
205         tag = Tag.UNKNOWN;
206         skipBeforeWhitespace();
207         value = start.getSubText(follow);
208     }
209 
210     private void skipBeforeWhitespace() {
211         while (!follow.satisfies(Character::isWhitespace) && follow.getChar() != -1) {
212             follow = follow.skip();
213         }
214     }
215 
216     public Token next() throws SyntaxError {
217         return new Token(follow);
218     }
219 
220     public String getValue() {
221         return value;
222     }
223 
224     public Tag getTag() {
225         return tag;
226     }
227 
228     public Position getStart() {
229         return start;
230     }
231 
232     public Position getFollow() {
233         return follow;
234     }
235 }
236 
237 public class Lab4 {
238     private static Token token;
239 
240     public static void main(String[] args) {
241         Scanner in = new Scanner(System.in);
242         in.useDelimiter("\\Z");
243         String text = in.next();
244 
245         try {
246             token = new Token(text);
247             do {
248                 printToken(
249                         token.getTag().toString(),
250                         token.getStart().toString(),
251                         token.getFollow().toString(),
252                         token.getValue()
253                 );
254                 token = token.next();
255 
256             } while (token.getTag() != Tag.END_OF_TEXT);
257         } catch (SyntaxError e) {
258             System.out.println(e.getMessage());
259         }
260     }
261 
262     private static void printToken(String tag, String start, String end, String value) {
263         System.out.printf("%s %s-%s: %s\n", tag, start, end, value);
264     }
265 }
```

Тестирование:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...
`aaaa`
!!!!
where
wher
-
->
a77
w
^D
OPERATION (1, 1)-(1, 7): `aaaa`
OPERATION (2, 1)-(2, 5): !!!!
KEYWORD (3, 1)-(3, 6): where
IDENT (4, 1)-(4, 5): wher
OPERATION (5, 1)-(5, 2): -
KEYWORD (6, 1)-(6, 3): ->
IDENT (7, 1)-(7, 4): a77
IDENT (8, 1)-(8, 2): w

Process finished with exit code 0
```

Вывод:

В результате выполнения лабораторной работы была реализована программа, реализующая лексический анализатор на объектно-ориентированном языке без применения средств автоматизации решения задачи лексического анализа.