## 2. Лексически анализ. Основни понятия и алгоритъм на лексическия анализ. Програмна структура на лексическия анализатор

### Цел на упражнението

Упражнението представя кратко въведение в теорията на лексическия анализ. Дадени са основните понятия и базовият алгоритъм на работа на лексическия анализатор. Обяснена е програмната структура на учебния лексически анализатор, най-важните методи и структури от данни, използвани за разпознаване на лексеми.

### 2.1. Лексически анализ - основни понятия

Основната задача на лексическия анализ е разпознаване на лексеми (tokens) във входната програма. Модулът, който изпълнява тази задача, се нарича лексически анализатор (или скенер). На входа на лексическия анализатор се получава поток от символи. Този поток представлява текста на входната програма, в който обикновено предварително са премахнати символите за нов ред. В резултат от работата на лексическия анализатор на неговия изход се получава поток от лексеми.

Лексемата се представя чрез структура от данни, която се състои от два основни елемента:

- тип на лексемата (напр. ключова дума, оператор, разделител, константа)

- стойност на лексемата, представляваща символ или символен низ (напр. if, >, {, 100)

Освен разпознаването на лексемите на етапа на лексическия анализ се решават още следните задачи:

- Откриване на лексически грешки във входната програма – невалидна лексема, невалидна числова, символна или низова константа

- Премахване на незначещи символи от входната програма – интервал, табулация

### 2.2. Лексика на входния език

Във входния език на учебния компилатор са дефинирани следните типове лексеми:

- идентификатори (имена на променливи и функции)

- ключови думи (напр. while, if, else, print, read, return)

- целочислени, булеви, символни и низови константи (напр. 10, true, '1', "5")

- едно и двусимволни оператори и разделители (напр. =, >, <, ==, >=, <=, {)

Лексиката на всеки език за програмиране се дефинира чрез азбука и граматика

#### 2.2.1. Азбука на входния език

Азбуката на входния език включва следните символи:

- малки и главни букви ('a'..'z', 'A'..'Z')

- цифри ('0'..'9')

- специални символи (например '+', '-', '%', ';')

### 2.2.2. Граматика на входния език

Граматиката се състои от правила, всяко от които има следния вид:

лява страна : дясна страна

Входният език на учебния компилатор е дефиниран чрез следната лексическа граматика:

**Ключови думи**

| | | |
|---|---|---|
| IF | : | 'if' |
| INT | : | 'int' |
| CHAR | : | 'char' |
| ELSE | : | 'else' |
| READ | : | 'read' |
| TRUE | : | 'true' |
| VOID | : | 'void' |
| FALSE | : | 'false' |
| WHILE | : | 'while' |
| PRINT | : | 'print' |
| LENGTH | : | 'length' |
| RETURN | : | 'return' |
| BOOLEAN | : | 'boolean' |
| PROGRAM | : | 'program' |

**Оператори**

| | | |
|---|---|---|
| PLUS | : | '+' |
| MINUS | : | '-' |

MUL : '*'

DIV : '/'

MOD : '%'

BECOMES : '='

**Релационни оператори**

NOT : '!'

EQUALS : '=='

NOTEQUALS : '!='

GREATER : '>'

LESS : '<'

GREATER_EQ : '>='

LESS_EQ : '<='

**Логически оператори**

AND : '&&'

OR : '||'

**Специални символи**

LSQUARE : '['

RSQUARE : ']'

LBRACKET : '{'

RBRACKET : '}'

LPAREN : '('

RPAREN : ')'

SEMICOLON : ';'

COMMA : ','

SINGLE_QUOTE : '"

DOUBLE_QUOTES : ""

ARROW : '->'

AT : '@'

**Целочислени, булеви и символни константи**

INT_LITERAL : '0' | [1-9]{1}[0-9]*

CHAR_LITERAL : any ascii character

BOOLEAN_LITERAL : 'true' | 'false'

**Идентификатор**

IDENTIFIER : [A-Za-z]{1}[A-Za-z0-9]*

В правилото за лексемата идентификатор (IDENTIFIER) е използван регулярен израз, който дефинира, че валиден идентификатор в езика започва с една буква (символ в диапазона ‘a-z’ или ‘A-Z’), следвана от произволен брой букви и/или цифри.

### 2.3. Работа на лексическия анализатор

Работата на лексическия анализатор се състои в последователно прочитане на всички символи от входния поток и причисляване на поредния прочетен символ към текущо формираната лексема.

По-долу е даден общият алгоритъм на работа на лексическия анализатор. Променливата currentChar съдържа поредния прочетен символ от входния текст.

```
(1)  while (currentChar != Source.EOF)
(2)  {
(3)      switch  (currentChar)  {
(4)          case ‘ ’: case ‘\t’:
(5)                  /* Премахване на незначещ символ интервал или табулация */
(6)          break;
(7)          case  ‘a’: case ‘b’: … case ‘z’ :
(8)                  /* Разпознаване на идентификатор или ключова дума */
```

```
(9)                 break;
(10)          case '0': case '1' : /* . . . */ case '9':
(11)                /* Разпознаване на целочислена константа */
(12)                 break;
(13)          case '\'':
(14)                /* Разпознаване на символна константа */
(15)                 break;
(16)          case '"':
(17)                /* Разпознаване на низова константа (символен низ) */
(18)                 break;
(19)           case '-': case '=': case '>': case '<': case '!': case '&': case '|': case '/':
(20)                /* Разпознаване на двусимволни оператори */
(21)                 break;
(22)           case '+': case '[': case ']': case '{': case '}': case '(': case ')': case ';': case '*':
case '%': case ',': case '@':
(23)                /* Разпознаване на едносимволни оператори и разделители */
(24)                break;
(25) };
```

Идентификаторите и ключовите думи започват с буква, поради което разпознаването им става в една и съща case клауза (редове 7 и 8). За разграничаване на идентификатори от ключови думи думите се записват предварително в подходяща структура от данни (напр. таблица или списък). При разпознаването на ключова дума тази структура се претърсва за откриване на низа на ключовата дума.

### 2.4. Програмна структура на учебния лексическия анализатор

На входа на лексическия анализатор се получава поток от символи. Този поток представлява входната програма, която е предварително преформатирана чрез премахване на символите за нов ред от входно-изходния интерфейс, реализиран в клас Source. В резултат лексическият анализатор получава входната програма като един символен низ.

В кода на лексическия анализатор лексемите са дефинирани в интерфейса Token (файл TokenImpl.java).

```
public interface Token<Type> {
   public Type getTokenType();
   public String getText();
   public int getPosition();
   public int getLine();
}
```

Този интерфейс описва тип на лексемата, низ на лексемата, номер на позиция и ред, на който се намира лексемата в текста на програмата.

### 2.4.1. Разпознаване на ключови думи

Ключовите думи се записват в структура keywords от тип HashSet с цел да бъдат разграничавани от идентификаторите.

```
   private static Set<String> keywords = new HashSet<>();
   static{
      keywords.add(IF.value);
      /* ToDo - Add all other keywords into the HashSet */


   }
```

Търсенето на ключова дума в структурата се изпълнява от метод isKeyword();

### 2.4.2. Разпознаване на оператори и разделители

Разпознаването на оператори и раздели се изпълнява от метода nextToken(). Този метод реализира основния алгоритъм на работа на учебния лексически анализатор.

```
(1)   public Token<TokenType> nextToken() {
(2)      currentChar = source.getCurrentChar();
(3)      line = source.getLineNumber();
(4)      position = source.getPosition() + 1;
(5)      while (currentChar != Source.EOF) {
(6)         switch (currentChar) {
(7)            //space and tabs
(8)            case ' ' : case '\t' : handleSpaceAndTabs(); continue;
(9)            //2 character operators
```

```
(10)        case '-' : return handleTwoCharOp('>', TokenType.MINUS,
TokenType.ARROW);
(11)        case '=' : /* ToDo handle operators '=' (BECOMES) and '==' (EQUALS) */
(12)        case '>' : /* ToDo handle operators '>' (GREATER) and '>=' (GREATER_EQ) */
(13)        case '<' : /* ToDo handle operators '<' (LESS) and '<=' (LESS_EQ) */
(14)        case '!' : /* ToDo handle operators '!' (NOT) and '!=' (NOTEQUALS) */
(15)        case '&' : /* ToDo handle operator '&&' (AND) or unknown symbol (OTHER)  */
(16)        case '|' : /* ToDo handle operator '|' (OR) or unknown symbol (OTHER) */
(17)        case '/' : return handleSlash();
(18)        case '\'': return handleCharLiteral();
(19)        case '"' : return handleStringLiteral();
(20)        //1 character operators
(21)        case '+' : return retTokenAndAdvance(TokenType.PLUS);
(22)        case '[' : /* ToDo handle operator '[' (LSQUARE) */
(23)        case ']' : /* ToDo handle operator ']' (RSQUARE) */
(24)        case '{' : /* ToDo handle operator '{' (LBRACKET) */
(25)        case '}' : /* ToDo handle operator '}' (RBRACKET) */
(26)        case '(' : /* ToDo handle operator '(' (LPAREN) */
(27)        case ')' : /* ToDo handle operator ')' (RPAREN) */
(28)        case ';' : /* ToDo handle operator ';' (SEMICOLON) */
(29)        case '*' : /* ToDo handle operator '*' (MUL) */
(30)        case '%' : /* ToDo handle operator '%' (MOD) */
(31)        case ',' : /* ToDo handle operator ',' (COMMA) */
(32)        case '@' : /* ToDo handle operator '@' (AT) */
(33)        default  :
(34)            if (isLetter(currentChar)) { return handleIdentifier(); }
(35)            if (isDigit(currentChar)) { return handleDigit(); }
(36)            return retTokenAndAdvance(TokenType.OTHER, currentChar + "");
```

```
(37)            }
(38)        }
(39)        return null;
(40)    }
```

Методът прочита пореден символ от потока на входната програма в променливата currentChar, проверява кой тип лексеми започва с този символ, след което започва разпознаване на съответния тип лексема. Прочетеният символ и следващите символи след него формират низа на лексемата.

Примери за едносимволни оператори са символите ‘+’, ‘*’, ‘%’. Едносимволни разделители са символите ‘(’,’)’, ’[’,’]’, ’{’,’, ’}’, ‘,’, ‘;’, ‘@’.

Примери за двусимволни оператори са ‘==’, ‘->’, ‘<=’, ‘>=’, ‘!=’, ‘||’.

- Разпознаване на едносимволни оператори и разделители

Например символ ‘+’ се разпознава като лексема с име PLUS със следното извикване на метода retTokenAndAdvance:

```
(21)            case '+' : return retTokenAndAdvance(TokenType.PLUS);
```

- Разпознаване на двусимволни оператори и разделители

При разпознаване на двусимволен оператор или разделител се проверява кой е следващият символ след поредния прочетен. В зависимост от следващия символ се формира съответен едно или двусимволен оператор или разделител.

Нека например в променливат currentChar е прочетен пореден символ ‘-’. Този символ може да формира едносимволен оператор ‘-’ или двусимволен разделител ‘->’. Затова се проверява следващият символ. Ако той е ‘>’, то се разпознава двусимволен оператор ‘->’ (ARROW). Ако символът е друг, тогава се разпознава едносимволен оператор ‘-’ (MINUS). Реализацията на този алгоритъм изглежда по следния начин:

```
(10)            case '-' : return handleTwoCharOp('>', TokenType.MINUS, TokenType.ARROW);
```

**2.4.3. Разпознаване на лексически грешки**

Лексическият анализатор разпознава следните типове грешки:

- Невалидна числена константа – това са константи със стойност над $2^{32}$
- Невалидна низова константа – това е напр. низ “aaa

Обработването на лексически грешки се реализира чрез генериране на изключение LexicalException.

**Контролни въпроси:**

1. Какви задачи се решават в етапа на лексическия анализ?

2. Какво означава понятието лексема?

3. Какво представлява входа и изхода на лексическия анализатор?

4. Определете лексемите в следния програмен фрагмент:

```
while ( i > 0) {
  a = a + 1;
}
```

**Задачи:**

1. Попълнете всички ключови думи в HashSet структурата keywords във файл TokenType.java на проекта.

2. Довършете фрагментите за разпознаване на едносимволни и двусимволни оператори и разделители във файл LexerImpl.java на проекта.

3. Създайте нов файл keywords.txt и го копирайте в директория resources на проекта. Запишете във файла всички ключови думи на езика. Стартирайте лексическия анализатор с този файл и проверете в изхода дали всяка ключова дума се разпознава правилно.

Упътване: Името на входния файл се задава в метода main:

```
public static void main(String[] args) throws IOException {
        Lexer<TokenType> lexer = new LexerImpl(new
SourceImpl("resources/keywords.txt"));
        System.out.println(CompilerTestHelper.getTokensAsString(lexer));
    }
```

4. Създайте нов файл operators.txt в директория resources на проекта. Запишете във файла всички оператори и разделители на езика. Стартирайте лексическия анализатор с този файл и проверете в изхода дали всички оператори и разделители се разпознават правилно.

5. В директория resources на проекта е дадена примерна входна програма (файл Fib.txt). Стартирайте лексическия анализатор с този файл. Проверете в изхода дали всички лексеми от програмата са разпознати правилно.

6. Да се променя входната програма и да се проверява изхода на лексическия анализатор.

**Допълнителни задачи:**

1. С помощта на инструмента Debug наблюдавайте четенето на символите от входния поток (т.е. стойностите на променливата currentChar по време на изпълнение).

2. С помощта на инструмента Debug наблюдавайте формирането на лексеми при изпълнение на лексическия анализатор.