ANÁLISE LÉXICA

1. Relacionar os tokens da linguagem;
2. Construir uma gramática léxica;
3. Reconhecer e classificar os tokens;
4. Retornar um objeto da classe Token;
5. Em caso de erro, consumir o caracter corrente e retornar o token ERRO;
6. Consumir separadores antes de cada token;
7. Retornar o token EOF quando atingir o final do arquivo;
8. Embutir todos os detalhes de acesso ao arquivo no meio externo.

Token	::=	Identifier | Integer-Literal | Operator | **;** | **:** | **:=** | **~** | **(** | **)** | eot
Identifier	::=	Letter | Identifier Letter | Identifier Digit
Integer-Literal	::=	Digit | Integer-Literal Digit
Operator	::=	**+** | **-** | ***** | **/** | **<** | **>** | **=** | ****
Separator	::=	Comment | space | eol
Comment	::=	**!** Graphic* eol

Token	::=	Letter (Letter | Digit)* | Digit Digit* |
		+ | - | * | / | < | > | = | \ |
		; | : (= | ε) | ~ | (|) | eot
Separator	::=	! Graphic* eol | space | eol

```
private byte scanToken () {
    switch (currentChar) {

    case 'a': case 'b': case 'c':
    ...       case 'y': case 'z':
        takeIt();
        while (isLetter(currentChar)
               || isDigit(currentChar))
            takeIt();
        return Token.IDENTIFIER;

    case '0': case '1': case '2':
    case '3': case '4': case '5':
    case '6': case '7': case '8':
    case '9':
        takeIt();
        while (isDigit(currentChar))
            takeIt();
        return Token.INTLITERAL;
```

Token ::=

Letter

(Letter | Digit)*

|

Digit

Digit*

```
case '+': case '-': case '*':
case '/': case '<': case '>':
case '=': case '\\':
   takeIt();
   return Token.OPERATOR;

case ';':
   takeIt();
   return Token.SEMICOLON;

case ':':
   takeIt();
   if (currentChar == '=') {
      takeIt();
      return Token.BECOMES;
   }
   else
      return Token.COLON;

case '~':
   takeIt();
   return Token.IS;

case '(':
   takeIt();
   return Token.LPAREN;
```

```
|
+|-|*|/|<|>|=|\

|
;

|
:

( =

|
ε )

|
~

|
(
```

```
    case ')':                                    |
        takeIt();                                )
        return Token.RPAREN;

    case '\000':                                 |
        return Token.EOT;                        eot

    default:
        report a lexical error
    }
}
```

```
private void scanSeparator () {
    switch (currentChar) {
    case '!': {
          takeIt();
          while (
             isGraphic(currentChar))
             takeIt();
             take('\n');
       }
       break;

    case ' ': case '\n':
       takeIt();
       break;
    }

}
```

Separator ::=

!

Graphic*

eol

|

space | eol

```
public class Scanner {

    private char currentChar = first source character;

    // Kind and spelling of the current token:
    private byte currentKind;
    private StringBuffer currentSpelling;

    private void take (char expectedChar) {
        if (currentChar == expectedChar) {
            currentSpelling.append(currentChar);
            currentChar = next source character;
        } else
            report a lexical error
    }

    private void takeIt () {
        currentSpelling.append(currentChar);
        currentChar = next source character;
    }
```

```
private boolean isDigit (char c) {
    ... // Returns true iff the character c is a digit.
}

private boolean isLetter (char c) {
    ... // Returns true iff the character c is a letter.
}

private boolean isGraphic (char c) {
    ... // Returns true iff the character c is a graphic.
}
```

```
    private byte scanToken () {
        ...  // As above.
    }

    private void scanSeparator () {
        ...  // As above.
    }

    public Token scan () {
        while (currentChar == '!'
               || currentChar == ' '
               || currentChar == '\n')
            scanSeparator();                          Separator*
        currentSpelling =
              new StringBuffer("");
        currentKind = scanToken();                    Token
        return new Token(currentKind,
                currentSpelling.toString());
    }
}
```

```
public class Token {

    public byte kind;
    public String spelling;

    public Token (byte kind, String spelling) {
        this.kind = kind; this.spelling = spelling;
        // If kind is IDENTIFIER and spelling matches one
        // of the keywords, change the token's kind accordingly:
        if (kind == IDENTIFIER)
            for (int k = BEGIN; k <= WHILE; k++)
                if (spelling.equals(spellings[k])) {
                    this.kind = k;  break;
                }
    }
```

```
    // Constants denoting different kinds of token:
    public final static byte
        IDENTIFIER = 0, INTLITERAL = 1, OPERATOR = 2,
        BEGIN = 3, CONST = 4, DO = 5, ELSE = 6, END = 7,
        IF = 8, IN = 9, LET = 10, THEN = 11, VAR = 12,
        WHILE = 13, SEMICOLON = 14, COLON = 15,
        BECOMES = 16, IS = 17, LPAREN = 18,
        RPAREN = 19, EOT = 20;

    // Spellings of different kinds of token (must correspond to the
    //  token kinds above):
    private final static String[] spellings = {
        "<identifier>", "<integer-literal>",
        "<operator>", "begin", "const", "do", "else",
        "end", "if", "in", "let", "then", "var",
        "while", ";", ":", ":=", "~", "(", ")", "<eot>"
    }
}
```