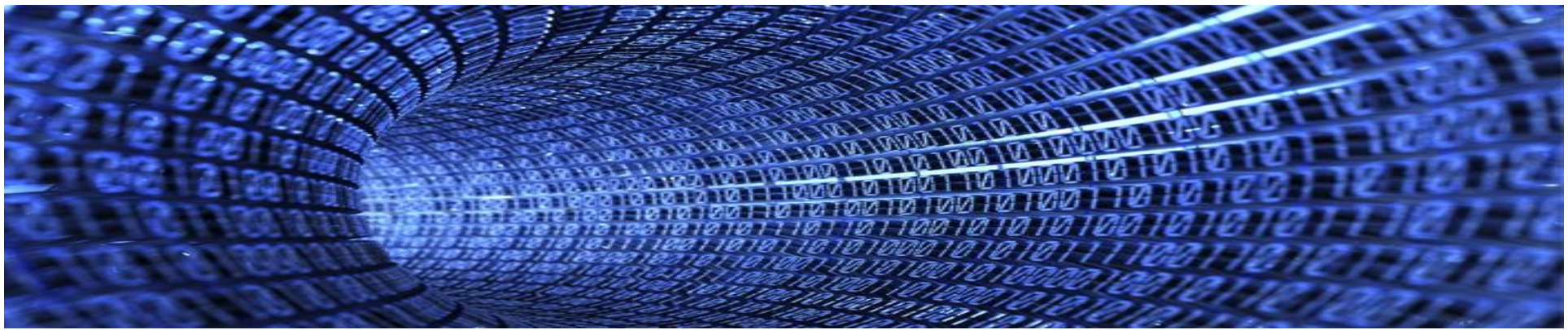
INSTITUTO MAUÁ DE TECNOLOGIA

MAUÁ

Curso de Engenharia de Computação
Linguagens Formais, Autômatos e Compiladores

JFlex – Gerador de analisadores léxicos em Java



Slides da disciplina ECM253 – Linguagens Formais, Autômatos e Compiladores
Curso de Engenharia de Computação
Instituto Mauá de Tecnologia – Escola de Engenharia Mauá
Prof. Marco Antonio Furlan de Souza
Marco.furlan@maua.br

O que é JFlex?

- **Conceitos**
  - JFlex é um **gerador** de **analisadores léxicos** (ou *scanners*) **para Java** e **escrito** em **Java** (baseado no Flex em C);
  - Ele toma como **entrada** uma **especificação** com um conjunto de **expressões regulares** e **ações** correspondentes. Depois **gera** um **programa** (***lexer***) que **lê** a **entrada**, **casa** a **entrada** com **expressões regulares** do arquivo de especificação e **executa** a **ação** especificada **quando** houver um **casamento** com a **expressão regular**;
  - É baseado em autômatos finitos determinísticos (DFA) – não **requer** *backtracking* e é de **rápida execução**;
  - Foi **projetado** para **trabalhar** com o **gerador** de **analisadores sintáticos CUP** (modificação do Yacc em C), mas pode ser utilizado com outros **geradores**, tais como **ANTLR** ou ainda isoladamente.

O que é JFlex?

- **Instalação e execução**
  - **Baixar** o arquivo compactado de http://jflex.de/download.html
  - **Descompactar** para alguma pasta. Os arquivos executáveis estão na pasta **`bin`** da distribuição:
    - No Windows: executar `jflex <opções> <arquivo de entrada>`
    - No Linux: executar `./jflex <opções> <arquivo de entrada>`
    - Ou adicionar a pasta **`bin`** ao caminho do sistema (PATH) ou adicionar um link no desktop e invocar a interface gráfica do programa.
  - **Documentação** para instalação e execução:
    - http://jflex.de/installing.html

Utilizando JFlex

- **Formato de um arquivo JFlex**
  - Um arquivo de entrada para o JFlex é um arquivo-texto com três seções, separadas por "%%":

```
Código do usuário
%%
Opções e declarações
%%
Regras léxicas
```

    - **Código do usuário**: o **código** aqui escrito é **copiado** da maneira como está **para** o início do código do **varredor**, antes da classe que implementa o varredor. É aqui que se escreve os comandos **`package`** e **`import`**.
    - **Opções e declarações**: são opções para personalizar o varredor a ser gerado, além de definir estados para o varredor e macro-definições;
    - **Regras léxicas**: para cada padrão a ser casado, define-se (ou não) uma regra a ser executada como resposta.

Utilizando JFlex

- **Exemplo 1**
  - Criar um analisador léxico Flex a partir do exemplo das Figuras 3.12 e 3.23 de (AHO; SETHI; LAM, 2007):

LEXEMAS	NOME DO TOKEN	VALOR DO ATRIBUTO
Qualquer *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Qualquer *id*	**id**	Apontador para entrada de tabela
Qualquer *number*	**number**	Apontador para entrada de tabela
`<`	**relop**	LT
`<=`	**relop**	LE
`=`	**relop**	EQ
`<>`	**relop**	NE
`>`	**relop**	GT
`>=`	**relop**	GE

FIGURA 3.12 Tokens, seus padrões e valores de atributo.

Utilizando JFlex

- **Exemplo 1**
  - Na primeira parte do arquivo (`exemplo1.jflex`), definiu-se apenas o pacote que o reconhecedor fará parte:

```
/**
 * Analisador léxico para expressões simples
 */
package exemplo1;
```

Utilizando JFlex

- **Exemplo 1**
  - Na segunda parte do arquivo, foram definidas opções e macros:

```
%class Lexer
%unicode
%standalone
%debug
%line
%column
%type Token

%eofval{
return new Token(Tag.EOF);
%eofval}

%eof{
System.out.println("Análise léxica terminada com sucesso!");
%eof}
```

Opções utilizadas

`class`: nome da classe do lexer;
`unicode`: suporte à Unicode;
`standalone`: cria método main() na classe;
`debug`: apresenta mensagens de depuração na execução;
`line` e **`column`**: disponibbiliza contadores de linha e coluna;
`type`: indica a classe de token a ser utilizada (veja mais à frente);
`eofval`: token a ser retornado quando se chegar ao fim de arquivo;
`eof`: ação a ser executado quando o varrimento terminar com êxito:

Utilizando JFlex

- **Exemplo 1**
  - Na segunda parte do arquivo, foram definidas opções e macros:

```
%{
// Macros
%}
delim    = [\ \t\n]
ws       = {delim}+
letter   = [A-Za-z]
digit    = [0-9]
id       = {letter}({letter}|{digit})*
number   = {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
```

`%{` e `%}` delimitam código Java

Macros não nomes que referenciarão expressões regulares a serem utilizadas na seção de regras.

Utilizando JFlex

- **Exemplo 1**
  - Na terceira parte do arquivo, foram definidas regras:

```
/* Regras e ações */
{ws}        { /* ignorar */ }
if          { return new Token(Tag.IF); }
then        { return new Token(Tag.THEN); }
else        { return new Token(Tag.ELSE); }
{id}        { return new Word(Tag.ID, yytext()); }
{number}    { return new Num(Double.parseDouble(yytext())); }
"<"         { return new Token(Tag.RELOP);}
"<="        { return new Token(Tag.RELOP);}
"="         { return new Token(Tag.RELOP);}
"<>"        { return new Token(Tag.RELOP);}
">"         { return new Token(Tag.RELOP);}
">="        { return new Token(Tag.RELOP);}
/* Qualquer outro - gerar erro */
.       { throw new Error("Illegal <" + yytext() +
          "(" + (int)(yytext().charAt(0)) + ")" + ">"); }
```

Veja a descrição das classes de `Token` nos próximos slides

Do lado esquerdo figuram aplicações de macros (entre `{` e `}`) ou valores literais (`if`, "`<`" etc).

Utilizando JFlex

- **Exemplo 1**
  - **Algumas variáveis e funções úteis do JFlex**
    - `yyline`: número da linha do código fonte durante a varredura. Depende da opção `%line`;
    - `yycolumn`: número da coluna do código fonte durante a varredura. Depende da opção `%column`;
    - `yytext()`: retorna o lexema (`String`) obtido pelo casamento da expressão regular com a entrada;
    - `yybegin(STATE)`: altera o estado do analisador para STATE. Existe um estado pré-definido denominado `YYINITIAL`, que representa o início da varredura da entrada. Pode-se adicionar quantos estados se quiser em um varredor.

Utilizando JFlex

- **Exemplo 1**
  - Na especificação do arquivo JFlex apresentado, foi utilizada a opção `%token Token`;
  - Isto significa que cabe ao **programador especificar** e utilizar uma **classe** para seus **tokens** – o **padrão** (se não usar `%token`) é utilizar o **formato** de **token** do programa **CUP** – gerador de analisadores sintáticos, também em Java;
  - Assim, foram definidas as classes a seguir, em Java, para implementar um token genérico e tokens específicos, armazenando seus atributos adicionais:
    - **`Token`**: classe para token geral
    - **`Num`**: classe específica para tokens referentes a números – herdada de `Token`;
    - **`Word`**: classe específica para tokens referentes a identificadores – herdada de `Token`;
    - **`Tag`**: classe auxiliar (na realidade uma classe de constantes públicas para classificar todos os tokens).

Utilizando JFlex

- **Exemplo 1**
  - Classe `Tag`

```
package exemplo1;

public class Tag {
    public final static int EOF = 256, NUMBER = 257, ID = 258,
            RELOP = 259, IF = 260, THEN = 261, ELSE = 262;
}
```

  - Define constantes que simplificarão a identificação dos tokens.

Utilizando JFlex

- **Exemplo 1**
  - Classe `Token:` armazena apenas a informação que classifica um token e possui, além do construtor, uma função que converte objeto desta classe em String, para fins de depuração.

```
package exemplo1;

public class Token {
    public final int tag;

    public Token(int t) {
        tag = t;
    }

    @Override
    public String toString() {
        return "<" + tag + ">";
    }
}
```

Utilizando JFlex

- **Exemplo 1**
  - Classe `Num`: além de armazenar a informação de que se trata de um token numérico, armazena seu lexema traduzido em um número real.

```
package exemplo1;

public class Num extends Token {
    public final double value;

    public Num(double v) {
        super(Tag.NUMBER);
        value = v;
    }

    @Override
    public String toString() {
        return "<" + this.tag + "," + this.value + ">";
    }
}
```

Utilizando JFlex

- **Exemplo 1**
  - Classe `Word`: além de armazenar a informação de que se trata de um token tipo identificador, também armazena seu lexema.

```
package exemplo1;

public class Word extends Token {
    public final String lexeme;

    public Word(int t, String s) {
        super(t);
        lexeme = new String(s);
    }

    @Override
    public String toString() {
        return "<" + this.tag + ",\"" + this.lexeme + "\">";
    }

}
```

Utilizando JFlex

- **Exemplo**
  - **Execução**
    - Em um terminal, a pasta em que o arquivo com definições JFlex se encontra, executar:

      ```
      jflex -d ./exemplo1 exemplo1.jflex
      ```

    - O parâmetro `-d` indica o diretório onde o código será gerado – neste caso, na pasta que tem o mesmo nome do pacote definido para o varredor. Ele criará o arquivo `Lexer.java`.
    - Depois, compilar o código do scanner:

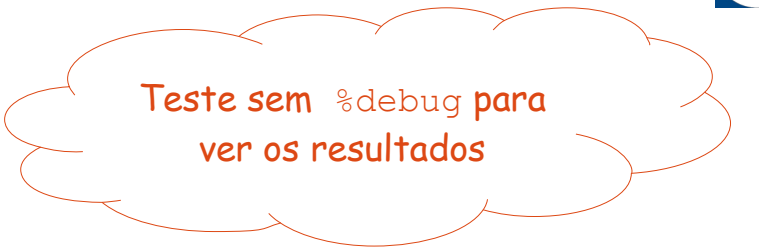
      ```
      javac exemplo1/Lexer.java
      ```

    - Este arquivo possui o método `main()`, logo, é possível executá-lo:

      ```
      java exemplo1.Lexer teste.input
      ```

Utilizando JFlex

- **Exemplo 1**
  - **Execução**
    - Quando o varredor é autônomo, apresenta-se apenas as ações executadas (causadas pela opção `%debug`):




```
    --
action [36] { /* ignorar */ }
line: 4 col: 2 match: --y--
action [40] { return new Word(Tag.ID, yytext()); }
line: 4 col: 3 match: -- --
action [36] { /* ignorar */ }
line: 4 col: 4 match: --=--
action [42] { return new Token(Tag.RELOP); }
line: 4 col: 5 match: -- --
action [36] { /* ignorar */ }
line: 4 col: 6 match: --6.02E23--
action [41] { return new Num(Double.parseDouble(yytext())); }
line: 4 col: 13 match: --
--
```

Utilizando JFlex

- **Exemplo 1**
  - **Automatização da construção do programa**
    - Para simplificar o processo de construção de programas Java, pode-se utilizar o **programa Ant**, que é uma ferramenta de construção cujos comandos são armazenados em um arquivo XML (normalmente com o nome `build.xml`);
    - Cada passo de construção pode ter um nome e realiza uma etapa. Por exemplo, no caso da construção de compiladores, pode-se ter uma etapa em que o analisador léxico é gerado, depois o sintático, depois as fontes são compiladas e, no final, as partes são empacotadas em um formato de distribuição;
    - Será exemplificado um arquivo Ant para criar um arquivo JAR para facilitar na execução do analisador léxico – ele invocará o Jflex, criará a classe do analisador, compilará todos os fontes e empacotará as classes compiladas em um JAR.

Utilizando JFlex

- **Exemplo 1**
  - **Automatização da construção do programa**
    - O projeto deste exemplo possui a organização descrita a seguir:

```
Exemplo01/
├── build.xml                 Arquivo de construção para o Ant
├── jflex
│   └── exemplo1.jflex        Fonte do analisador léxico
├── src
│   └── exemplo1
│       ├── Num.java          Classe para token numérico
│       ├── Tag.java          Classe de tag - número do token
│       ├── Token.java        Classe de token genérico
│       └── Word.java         Classe de token para cadeia de caracteres
├── teste.input               Arquivo para testar o analisador
└── tools
    └── jflex-1.6.1.jar       Programa JFlex
```

Utilizando JFlex

- **Exemplo 1**
  - **Automatização da construção do programa**
    - Código de `build.xml`:

```
<project name="Exemplo1" default="dist" basedir=".">
    <description>
        Script Ant para compilar projetos com JFlex
    </description>
    <!-- Propriedades globais -->
    <property name="jflexFile" value="exemplo1.jflex" />
    <property name="mainClass" value="exemplo1.Lexer" />
    <property name="jarFile" value="exemplo1.jar" />
    <property name="src" location="src" />
    <property name="tools" location="tools" />
    <property name="jflex" location="jflex" />
    <property name="build" location="build" />
    <property name="dist" location="dist" />
    <taskdef  name="jflex" classname="jflex.anttask.JFlexTask"
              classpath="${tools}/jflex-1.6.1.jar" />
```

Utilizando JFlex

- **Exemplo 1**
  - **Automatização da construção do programa**
    - Código de `build.xml` (cont.):

```
<target name="init">
    <!-- Criar o diretório build, de construção se ele não existir -->
    <mkdir dir="${build}" />
</target>
<target name="compile" depends="init" description="compilar as fontes">
    <!-- Executar jflex sobre o arquivo JFlex definido -->
    <jflex file="${jflex}/${jflexFile}" destdir="${src}" />
    <!-- Compilar o código Java em ${src} para ${build} -->
    <javac srcdir="${src}" destdir="${build}" />
</target>
<target name="dist" depends="compile" description="gerar a distribuição">
    <!-- Criar o diretório dist se ele não existir -->
    <mkdir dir="${dist}" />
    <!-- Empacotar o que existir em ${build} em  um arquivo .jar -->
    <jar jarfile="${dist}/${jarFile}" basedir="${build}">
        <manifest>
            <attribute name="Main-Class" value="${mainClass}" />
        </manifest>
    </jar>
</target>
```

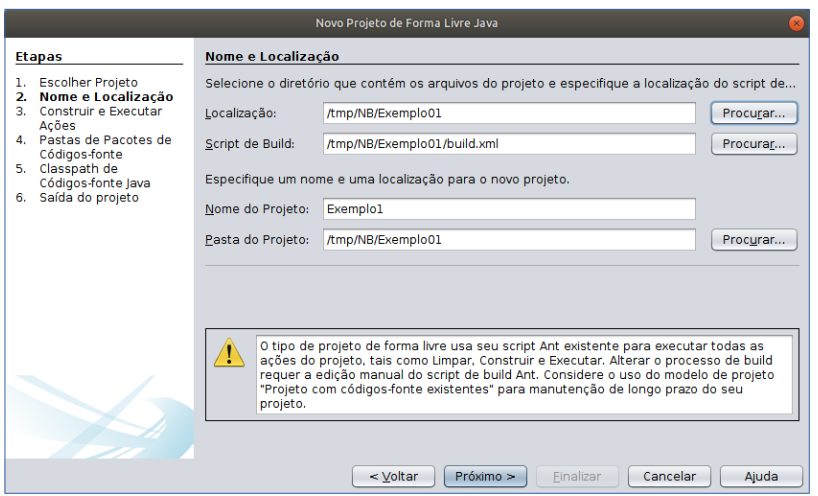
Utilizando JFlex

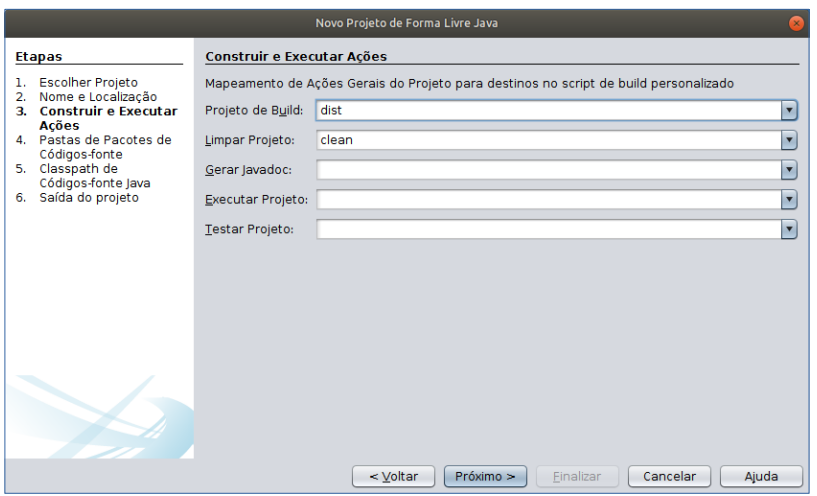
- **Exemplo 1**
  - **Automatização da construção do programa**
    - Código de `build.xml` (cont.):

```
    <target name="clean" description="clean up">
        <!-- Apaga o que foi gerado em ${build} e ${dist} -->
        <delete dir="${build}" />
        <delete dir="${dist}" />
    </target>
</project>
```

Utilizando JFlex

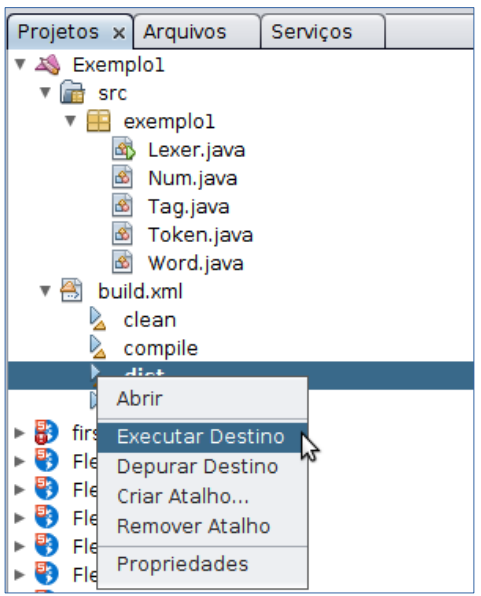
- **Exemplo 1**
  - **Automatização da construção do programa**
    - No Netbeans:
      - Definir no projeto uma estrutura como a apresentada no slide 19 (pastas e arquivos);
      - No Netbeans: Arquivo → Novo Projeto → Projeto de Forma Livre Java:





Utilizando JFlex

- **Exemplo 1**
  - **Automatização da construção do programa**
    - No Netbeans:
      - Para executar um “alvo”:



Utilizando JFlex

- **Exemplo 1**
  - **Automatização da construção do programa**
    - Para executar seu analisador (ele está na pasta `dist`):

```
java -jar exemplo1.jar ../teste.input
```

Utilizando JFlex

- **Exemplo 2**
  - Neste exemplo, o **varredor** a ser gerado **não** será **autônomo**: será **chamado** por um **código Java** contendo o **método** `main()`;
  - O **objetivo** deste exemplo é **isolar textos** dentro de **tags** `<p>` e `</p>` em documentos **HTML5**;

```
Exemplo02
├── build.xml
├── jflex
│   └── exemplo2.jflex
├── src
│   └── exemplo2
│       ├── Main.java
│       ├── Tag.java
│       ├── Token.java
│       └── Word.java
├── teste.html
└── tools
    └── jflex-1.6.1.jar
```

Main.java contém o programa principal que invocará a função yylex() do varredor para obter um próximo token da entrada.

Utilizando JFlex

- **Exemplo 2**
  - Programa principal

```
package exemplo2;

public class Main {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Uso : java -jar <arquivo>");
        } else {
            Lexer scanner = null;
            try {
                java.io.FileInputStream stream = new
                    java.io.FileInputStream(args[0]);
                java.io.Reader reader = new java.io.InputStreamReader(stream);
                scanner = new Lexer(reader);
                Token token = scanner.yylex();
                while (token.tag != Tag.EOF) {
                    if (token instanceof Word) {
                        System.out.println(((Word) token).lexeme);
                    }
                    token = scanner.yylex();
                }
```

Utilizando JFlex

- **Exemplo 2**
  - Programa principal (cont.)

```
            } catch (java.io.FileNotFoundException e) {
                System.out.println("Arquivo não encontrado : \"" + args[0] + "\"");
            } catch (java.io.IOException e) {
                System.out.println("Erro de E/S durante a varredura : \"" + args[0]
                    + "\"");
                System.out.println(e);
            } catch (Exception e) {
                System.out.println("Exceção não esperada:");
                e.printStackTrace();
            }
        }
    }
}
```

Utilizando JFlex

- **Exemplo 2**
  - Especificação do varredor

```
/**
 * Analisador léxico para tags
 */
package exemplo2;

%%

%class Lexer
%unicode
%type Token
%eofval{
return new Token(Tag.EOF);
%eofval}
%line
%column
%eof{
System.out.println("Análise léxica terminada com sucesso!");
%eof}

%{
private StringBuffer buffer = new StringBuffer();
%}
```

Utilizando JFlex

- **Exemplo 2**
  - Especificação do varredor (cont.)

```
delim   = [\ \t\n]
ws      = {delim}+

%xstate PAR
%xstate END

%%
<YYINITIAL> {
"<p>"    { yybegin(PAR); buffer.setLength(0);}
{ws}     { }
.    { }
}
<PAR> {
"</p>"       { yybegin(YYINITIAL);
               return new Word(Tag.PAR_TEXT, buffer.toString()); }
.|{delim}    { buffer.append(yytext()); }
}
```

Utilizando JFlex

- **Exemplo 2**
  - Especificação do varredor (cont.)
    - Para reconhecer o texto dentro dos tags, foi utilizado o conceito de **estado** do **varredor**;
    - **Estado** é um **símbolo** que se **estiver presente**, pode-se **executar ações específicas** para este estado;
    - **Declara-se** um estado com `%state` ou `%xstate` (exclusivo);
    - Depois, define-se que o **varredor entrará** em um certo **estado** `S` com `yybegin(S)`;
    - As regras e ações que devem ser executadas nesse estado são rotuladas com `<S>`;
    - O único estado pré-determinado é `YYINITIAL`, padrão do varredor;
    - No exemplo apresentado, utiliza-se `StringBuffer` para compor a cadeia que vai sendo descoberta durante o varrimento.

Referências Bibliográficas

- AHO, A. V.; SETHI, R.; LAM, M. S. **Compiladores: princípios, técnicas e ferramentas**. 2. ed. [s.l.] Pearson, 2007.
- **JFlex – Documentation**. Disponível em: <http://jflex.de/docu.html>. Acesso em: 13 set. 2015.