Aula prática 1

Análise Léxica

José Romildo Malaquias*

Resumo

Nesta aula o aluno deverá familiarizar-se com o gerador de analisador léxico **JFlex** e, usá-lo para desenvolver analisadores léxicos para linguagens simples.

Sumário

1 Introdução

A **análise léxica** é a primeira etapa do processo de compilação e seu objetivo é dividir o código fonte em símbolos léxicos, preparado-o para a análise sintática. Neste processo pode-se destacar três atividades como fundamentais:

- extração e classificação dos símbolos léxicos que compõem o programa fonte,
- eliminação de caracteres brancos, como espaços em branco, tabulação e mudanças de linha, e comentários, e

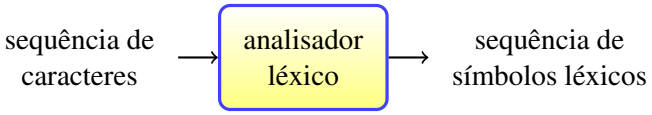
*romildo@iceb.ufop.br

- recuperação de erros léxicos, gerados por sequências de caracteres que não formam símbolos léxicos.

Símbolos léxicos, ou ***tokens***, são as palavras e sinais de pontuação utilizados para expressar a estrutura de um programa em um texto.

O **analisador léxico**, ou *scanner*, é um módulo do compilador que tem como entrada uma sequência de caracteres (texto do programa), produzindo na saída uma sequência de símbolos léxicos. O analisador léxico atua como uma interface entre o texto de entrada e o analisador sintático.

Figura 1: Analisador léxico.



As formas mais comuns de símbolos léxicos são:

identificadores palavras utilizadas para nomear entidades do programa, como variáveis, funções, métodos, classes, módulos, etc.

literais sequência de caracteres que representa uma constante, como um número inteiro, um número em ponto flutuante, um caracter, uma *string*, um valor verdade (verdadeiro ou falso), etc.

palavras chaves palavras usadas para expressar estruturas da linguagem, como comandos condicionais, comandos de repetição, etc. Geralmente são reservadas, não podendo ser utilizadas como identificadores.

sinais de pontuação sequências de caracteres que auxiliam na construção das estruturas do programa, como por exemplo servindo de separador de expressões em uma lista de expressões.

2 Analisador léxico *ad hoc*

Quando a estrutura léxica de uma linguagem não é muito complexa, o analisador léxico pode ser facilmente escrito *à mão*. O programa deve analisar a sequência de caracteres da entrada, agrupando-os para formar os tokens de acordo com a linguagem que está sendo implementada.

3 Especificação de símbolos léxicos

A estrutura léxica das linguagens de programação geralmente são simples o suficiente para a utilização de expressões regulares em sua especificação.

3.1 Cadeias e linguagens

Alfabeto é um conjunto finito não vazio de **símbolos**. São exemplos de símbolos letras, dígitos e sinais de pontuação. Exemplos de alfabeto:

- alfabeto binário: $\{0, 1\}$
- ASCII
- Unicode

Uma **cadeia** em um alfabeto é uma sequência finita de símbolos desse alfabeto. Por exemplo, 1000101 é uma cadeia no alfabeto binário.

O **tamanho** de uma cadeia s, denotado por $|s|$, é o número de símbolos em s. Por exemplo, $|1000101| = 7$.

A **cadeia vazia**, representada por ϵ, é a cadeia de tamanho zero.

Um **prefixo** de uma cadeia s é qualquer cadeia obtida pela remoção de zero ou mais símbolos do final de s. Por exemplo, ama, $amar$, a e ϵ são prefixos da cadeia $amarelo$.

Um **sufixo** de uma cadeia s é qualquer cadeia obtida pela remoção de zero ou mais símbolos do início de s. Por exemplo, elo, $amarelo$, o e ϵ são sufixos da cadeia $amarelo$.

Uma **subcadeia** de uma cadeia s é qualquer cadeia obtida removendo-se qualquer prefixo e qualquer sufixo de s. Por exemplo, are, mar, $relo$ e ϵ são subcadeias da cadeia $amarelo$.

Um prefixo, um sufixo ou uma subcadeia de uma cadeia s é dito **próprio** se não for a cadeia vazia ϵ ou a própria cadeia s.

Uma **subsequência** de uma cadeia s é qualquer cadeia formada pela exclusão de zero ou mais símbolos (não necessariamente consecutivos) de s. Por exemplo, $aaeo$ é uma subsequência da cadeia $amarelo$.

A **concatenação** xy (também escrita como $x \cdot y$) de duas cadeias x e y é a cadeia formada pelo acréscimo de y ao final de x. Por exemplo, $verde \cdot amarelo = verdeamarelo$. A cadeia vazia é o elemento neutro da concatenação: para qualquer cadeia s, $\epsilon s = s\epsilon = s$. A concatenação é associativa: para quaisquer cadeias r, s e t, $(rs)t = r(st) = rst$. Porém a concatenação não é comutativa.

3.2 Linguagens

Uma **linguagem** é um conjunto contável de cadeias de algum alfabeto. Exemplos de linguagens:

- o conjunto vazio, ϕ
- o conjunto unitário contendo apenas a cadeia vazia, $\{\epsilon\}$
- o conjunto dos números binários de 3 dígitos, $\{000, 001, 010, 011, 100, 101, 110, 111\}$
- o conjunto de todos os programas Java sintaticamente bem formados
- o conjunto de todas as sentenças portuguesas gramaticalmente corretas

As operações sobre linguagens mais importantes para a análise léxica são união, concatenação e fechamento.

A união de duas linguagens L e M é dada por

$$L \cup M = \{s | s \in L \text{ ou } s \in M\}$$

A concatenação de duas linguagens L e M é dada por

$$LM = \{st | s \in L \text{ e } t \in M\}$$

Uma potência de uma linguagem L é dada por

$$L^n = \begin{cases} \{\epsilon\} & \text{se } n = 0 \\ LL^{i-1} & \text{se } n > 0 \end{cases}$$

O fechamento Kleene de uma linguagem L é dado por

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

O fechamento positivo de uma linguagem L é dado por

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

3.3 Expressões regulares

Uma **expressão regular** (também conhecida como **padrão**) descreve um conjunto de cadeias de símbolos de forma concisa, sem precisar listar todos os elementos do conjunto. Por exemplo, o conjunto formado pelas cadeias *Handel*, *Händel* e *Haendel* pode ser descrito pelo padrão `H(ä|ae?)ndel`.

Segundo a teoria das linguagens formais, uma expressão regular sobre um conjunto finito Σ de símbolos (chamado de **alfabeto**) denota uma linguagem regular sobre Σ:

- o **conjunto vazio** ϕ é uma expressão regular que denota a linguagem vazia $\{\}$
- a **cadeia vazia** ϵ é uma expressão regular que denota a linguagem $\{\epsilon\}$
- o **literal** $a \in \Sigma$ é uma expressão regular que denota a linguagem $\{a\}$
- se R e S são expressões regulares, então RS, também escrito como $R \cdot S$, é uma expressão regular que denota a linguagem $\{\alpha\beta | \alpha \in L(R) \wedge \beta \in L(S)\}$, cujas cadeias são formadas pela **concatenação** de uma cadeia de $L(R)$ com uma cadeia de $L(S)$
- se R e S são expressões regulares, então $R|S$ é uma expressão regular que denota a linguagem $L(R) \cup L(S)$
- se R é uma expressão regular, então $R\star$ é uma expressão regular que denota a linguagem $\{\epsilon\} \cup L(R) \cup L(RR) \cup L(RRR) \cup \cdots$

4 Geradores de analisadores léxicos

Os geradores de analisadores léxicos são ferramentas que tem como entrada uma especificação da estrutura léxica de uma linguagem (na forma de um arquivo texto), e produzem um analisador léxico correspondente à especificação.

5 JFlex

JFLex (`http://jflex.de/`) é um gerador de analisador léxico escrito em Java que gera código em Java. Ele é distribuído usando o licença BSD e está disponível em `http://jflex.de/download.html`. O seu manual pode ser obtido em `http://jflex.de/manual.pdf`.

5.1 Instalação

Sendo uma aplicação Java, JFlex necessita de uma máquina virtual Java (JRE – Java Runtime Environment) para ser executado, e de um compilador Java para compilar as classes geradas (JDK – Java Development Kit). Assim certifique-se primeiro de que uma máquina virtual de Java já está instalada.

JFlex deve ser executado em qualquer plataforma que suporte um JRE / JDK 1.5 ou superior.

Existe um plugin para o Eclipse. Caso opte por instá-lo, não será necessáro instalar o JFlex separadamente.

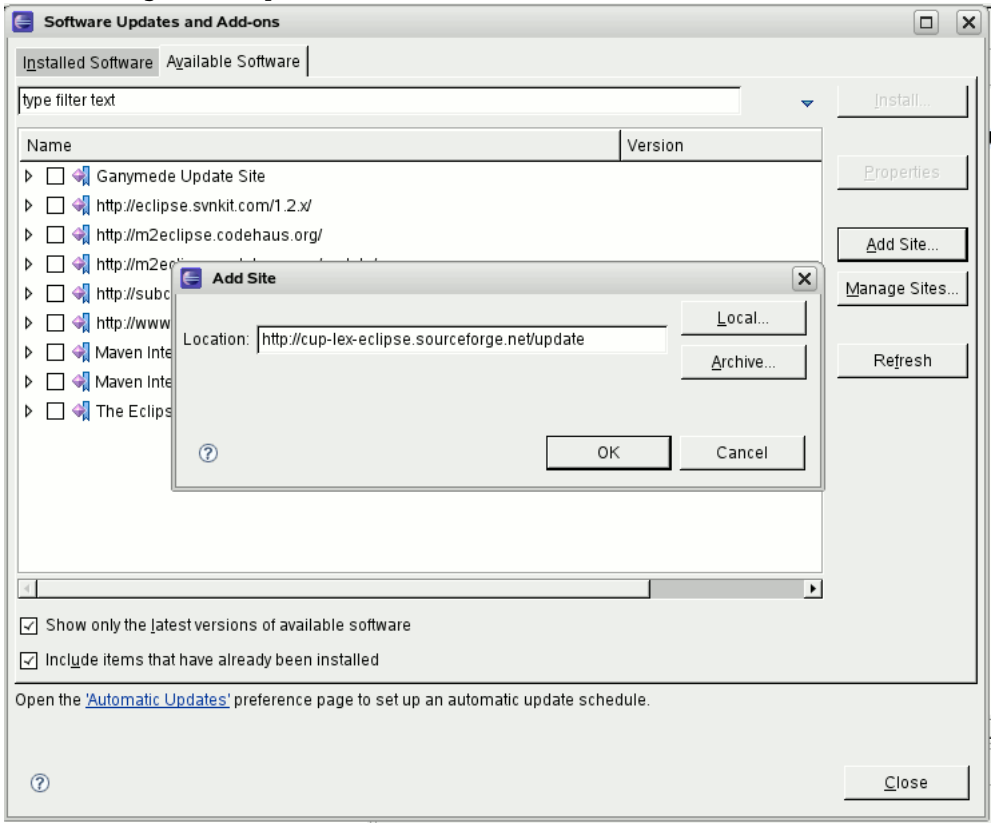
5.1.1 Instalação do plugin para Eclipse

CLE (`http://cup-lex-eclipse.sourceforge.net/`) é um *plugin* do Eclipse para editar arquivos de especificação de gramática utilizados pelo Java CUP (gerador de analisador sintático) e JFlex (gerador de analisador léxico).

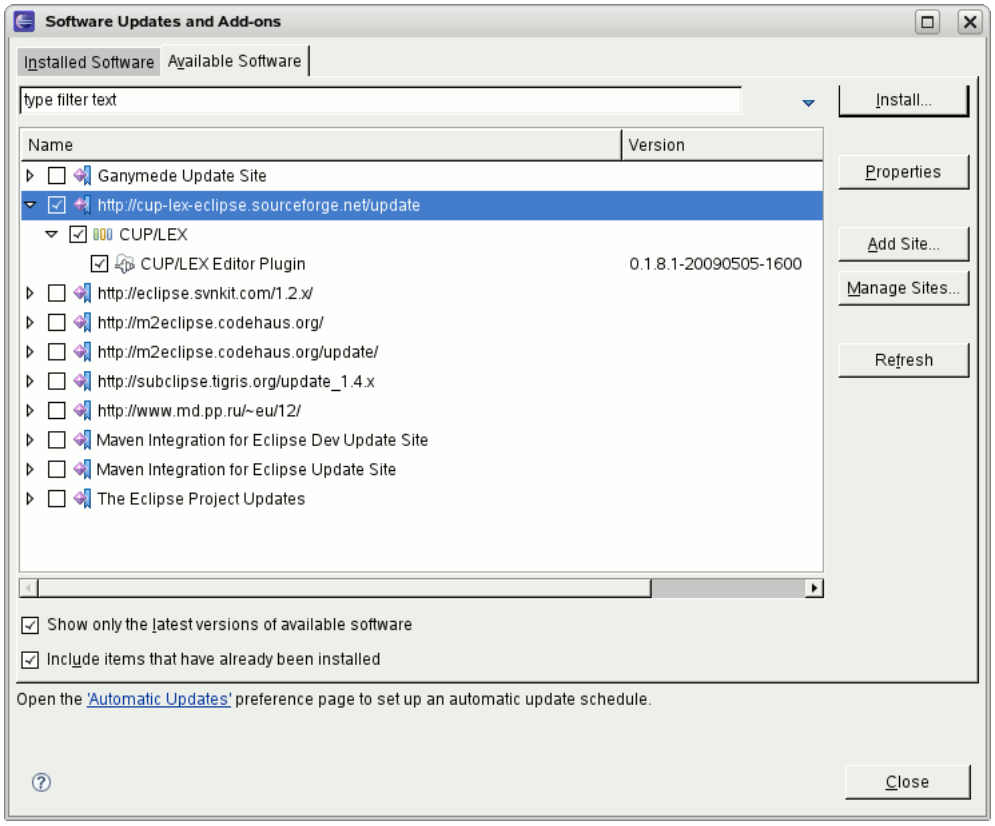
No momento em que este texto foi escrito este plugin encontrava-se desatualizado, não usando as versões mais recentes do JFlex e do Java CUP. Por este motivo o seu uso é desencorajado.

As instruções de instalação do CLE podem ser encontradas em `http://cup-lex-eclipse.sourceforge.net/installation.html`. Basicamente siga os seguintes passos:

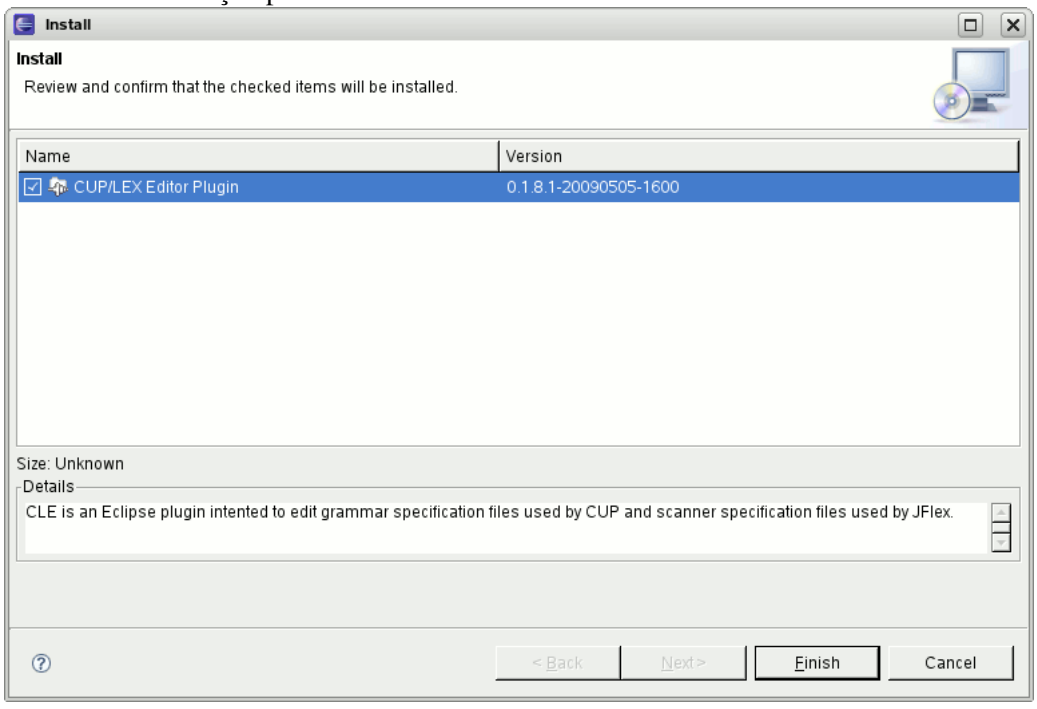
1. A partir do menu do Eclipse escolha *Ajuda* | *Software Updates*
2. Alterne para a guia *Software Disponível*, se necessário
3. Acione *Adicionar Site*
4. Preencha a caixa de diálogo com o local de atualização do CLE: `http://cup-lex-eclipse.sourceforge.net/update`



5. Na janela de atualização, agora você vai ver a categoria CUP/LEX; selecione-a e em seguida, pressione *Instalar*.

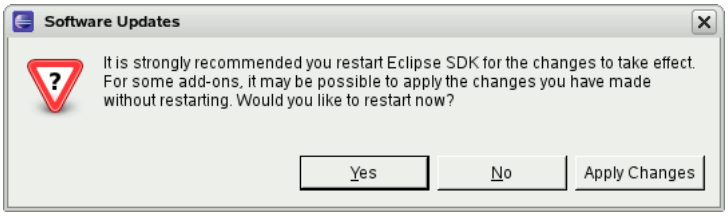


6. Confirme a instalação pressionando *Concluir*.



7. Aguarde o Gerenciador de Atualizações concluir a instalação do plugin.

8. Pressione *Sim* para reiniciar o Eclipse.



9. Isso é tudo!

5.1.2 Instalação no Windows (sem o plugin do Eclipse)

Para instalar JFlex no Windows, siga os passos seguintes:

1. Descompacte o arquivo *zip* disponibilizado em `http://jflex.de/download.html` em um diretório de sua preferência. Vamos assumir que este diretório é `C:\`, e que a versão do JFlex é a última disponível (1.5.1, neste momento: `http://jflex.de/jflex-1.5.1.zip`). Assim o JFlex estará disponível em `C:\jflex-1.5.1`.

2. Edite o arquivo `bin\jflex.bat` de forma que a variável de ambiente `JFLEX_HOME` contenha o diretório onde o JFlex foi instalado. No exemplo este diretório é `C:\jflex-1.5.1`.

3. Inclua o diretório `bin\` do JFlex (no exemplo, `C:\jflex-1.5.1\bin`) na variável de ambiente `PATH`.

5.1.3 Instalação no Linux (sem o plugin do Eclipse)

Sugiro que se utilize o programa de gerenciamento de pacotes de sua distribuição Linux para instalar o JFlex. Caso não haja um pacote disponível para a sua distribuição, siga as instruções em `http://jflex.de/installing.html`.

5.1.4 Instalação como ferramenta externa no Eclipse (sem o plugin)

O JFlex pode ser integrado ao Eclipse, um ambiente de desenvolvimento integrado largamente utilizado para o desenvolvimento de aplicações Java. Para tanto siga os passos segintes. Assumiremos que a versão do Eclipse é a versão 3.5.2.

1. Abra a janela de configuração de ferramentas externas acessando o menu *Run -> External Tools -> External Tools Configurations...*

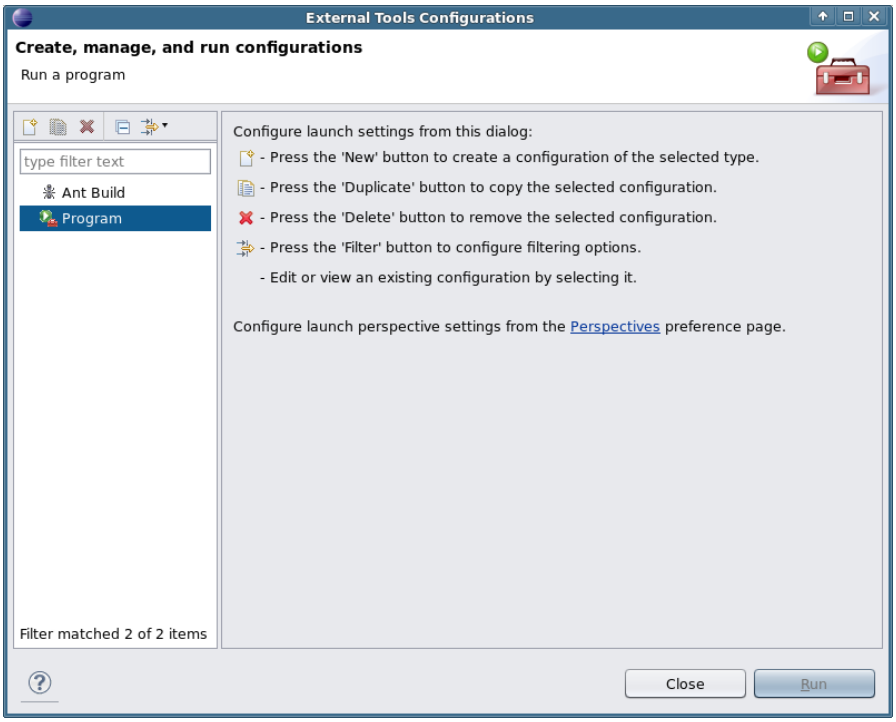
2. Na janela *External Tools Configuration* Clique no botão *New Launch Configuration* para criar uma nova configuração. Veja a figura 2.

3. Modifique o nome da configuração para `JFlex`.

4. Na aba *Main* preencha os seguintes campos (veja a figura 3):

   - Coloque o caminho para o arquivo de execução do JFlex no campo *Location*. No exemplo do Windows o caminho é `C:\jflex-1.5.1\bin\jflex.bat`. No linux o caminho provavelmente é `/usr/bin/jflex`.
   - Em *Working Directory* coloque `${container_loc}`. A variável `${container_loc}` corresponde ao caminho absoluto no sistema de arquivos do pai (um diretório ou um projeto) do recurso selecionado. No caso o recurso selecionado é o arquivo de especificação do JFlex a ser compilado.

Figura 2: Criando uma nova configuração de ferramenta externa.



- Em *Arguments* coloque `${resource_name}`. A variável `${resource_name}` corresponde ao nome do recurso selecionado.

5. Na aba *Refresh* marque as opções seguintes (veja a figura 4):
   - Marque a opção *Refresh resources upon completion.*
   - Marque a opção *The project containing the selected resource.*
6. Clique no botão *Apply* e depois no botão *Close.*

5.2 Executando o JFlex

5.2.1 Linha de comando

O JFlex pode ser executado na linha de comando com:

```
jflex <opções> <arquivos de entrada>
```

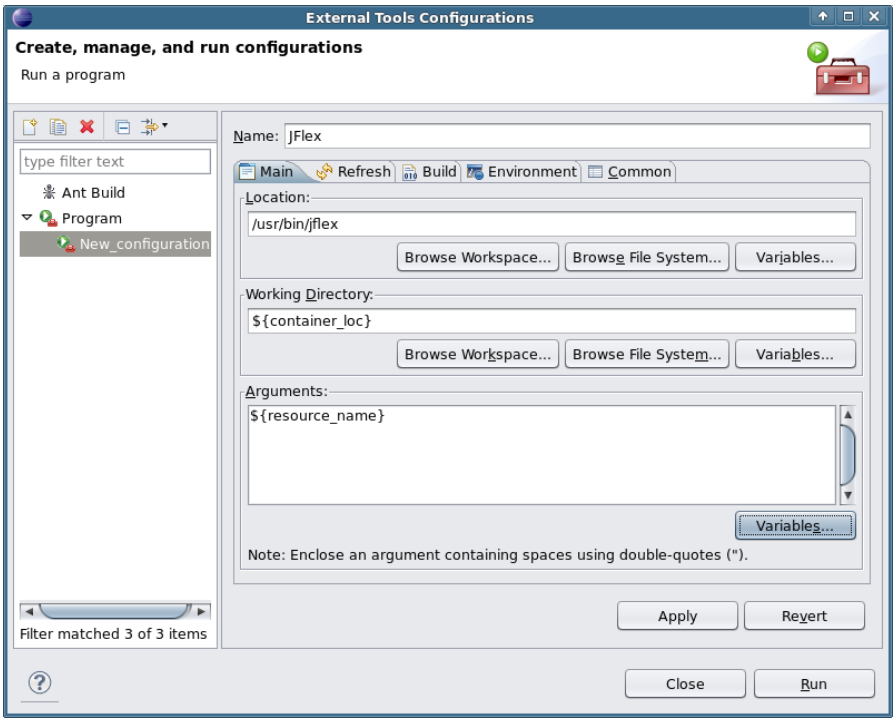
A opção `--help` exibe uma mensagem de auxílio explicando as opções e o uso do JFlex.

```
$ jflex --help

Usage: jflex <options> <input-files>

Where <options> can be one or more of
-d <directory>    write generated file to <directory>
--skel <file>     use external skeleton <file>
--switch             (DEPRECATED - will be removed in JFlex 1.6)
--table              (DEPRECATED - will be removed in JFlex 1.6)
--pack            set default code generation method (default)
```

Figura 3: Aba *Main* da configuração do JFlex no Eclipse.

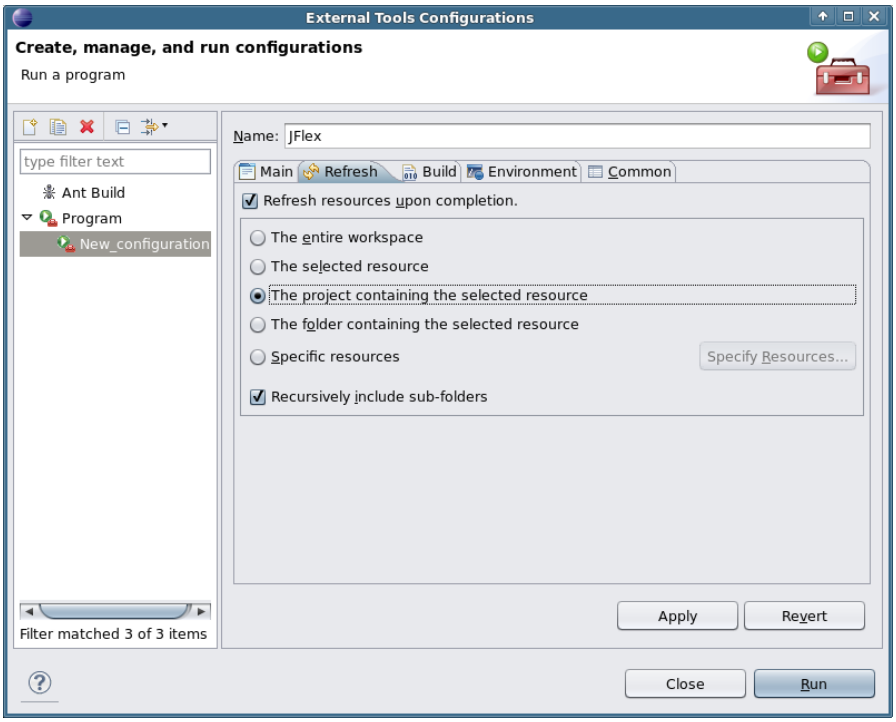



```
--jlex              strict JLex compatibility
--legacydot         dot (.) metachar matches [^\n] instead of
                    [^\n\r\u000B\u000C\u0085\u2028\u2029]
--inputstreamctor     include a scanner constructor taking InputStream (default)
--noinputstreamctor   don't include a scanner constructor taking InputStream
--nomin             skip minimization step
--nobak             don't create backup files
--dump              display transition tables
--dot               write graphviz .dot files for the generated automata (alpha)
--verbose
-v                  display generation progress messages (default)
--quiet
-q                  display errors only
--time              display generation time statistics
--version           print the version number of this copy of jflex
--info              print system + JDK information
--uniprops <ver>    print all supported properties for Unicode version <ver>
--help
-h                  print this message

This is JFlex 1.5.1
Have a nice day!
```

Consulte o manual para conhecer mais detalhadamente as opções que o JFlex aceita.

Figura 4: Aba *Refresh* da configuração do JFlex no Eclipse.



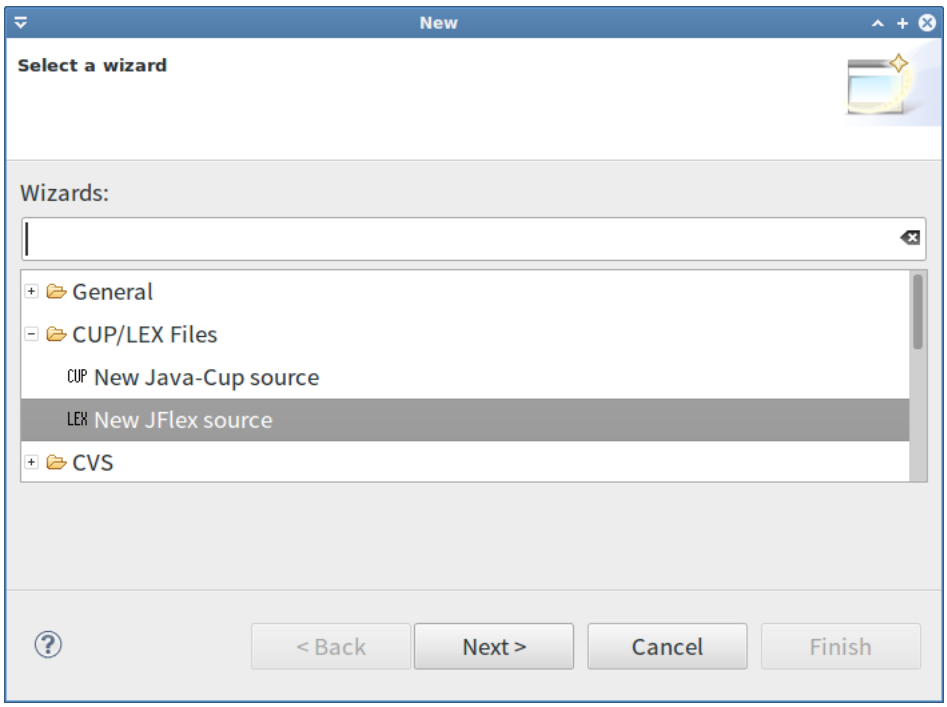
5.2.2 No Eclipse sem o plugin

Para compilar uma especificação, abra o arquivo contendo a especificação, vá à janela de configuração de ferramentas externas acessando o menu *Run -> External Tools -> External Tools Configurations...*, selecione a ferramenta JFlex, e clique no botão *Run*. Da próxima vez não será mais necessário abrir esta janela, pois uma entrada do JFlex é adicionada ao menu: *Run -> External Tools -> JFlex*.

5.2.3 No Eclipse com o plugin

Usando o plugin CLE é muito simples compilar uma especificação do JFlex: basta salvar o arquivo após alterar o seu conteúdo.

Para criar uma nova especificação do JFLex existe um assistente disponível no menu *File -> New -> Other -> CUP/LEX Files -> New JFlex source.*



Tarefa 1

Verifique a configuração do JFlex e do Eclipse em seu computador.

5.3 Exemplo: analisador simples

Uma especificação léxica é formada por três seções, separadas por %%:

1. **Código do usuário**. É inserido no início do arquivo gerado pelo JFlex. Tipicamente colocamos declaração de pacote e declaração de importação nesta seção.

2. **Opções e declarações**. É um conjunto de

   - opções que permitem configurar como o analisador léxico será gerado,
   - declarações de estados, e
   - declarações de macros

   Cada opção deve começar com o caracter %, colocado na primeira coluna da linha.

3. **Regras léxicas**. É um conjunto de regras, onde cada regra é formada por uma expressão regular e uma açãoo. A expressão regular identifica uma classe de símbolos léxicos, e a ação é um trecho de código Java que é executado quando um símbolo é formado durante a análise léxica utilizando a expressão regular correspondente. Caso haja conflito entre as regras, decidie-se pela regra que produzir a maior cadeia. Se ainda assim persistir o conflito, dá-se preferência à regra que aparece primeiro na especificação. O texto da entrada que casa com a expressão regular pode ser acessado pelo método `yytext`, que é incluído na classe gerada.

A listagem 1 contém uma especificação léxica bem simples.

Listagem 1: Exemplo de especificação léxica.

```
%%

%integer

%%

[a-z][a-z0-9]* { return 1; }
[0-9]+         { return 2; }
[ \t\n\r]+     { /* do nnothing */ }
.              { System.out.printf("error: unexpected char |%s|\n",
                                   yytext());
               }
```

1. Identificadores são formados por uma sequência de letras minúsculas e dígitos decimais, começando por uma letra. O tipo do símbolo `1`.
2. Números naturais são formados por uma sequência de um ou mais dígitos decimais. O tipo do símbolo é `2`.
3. Brancos (espaços, tabulação horizontal, mudança de linha) são ignorados.
4. Qualquer caracter da entrada que não formar símbolo léxico (identificador ou número inteiro) ou for ignorado (brancos) gera uma mensagem de erro. É importante que esta seja a última regra, pois ela casa com qualquer caracter da entrada (exceto mudança de linha).

Por default, JFlex gera uma classe chamada `Yylex`. A opção `%class` permite especificar o nome da classe desejada. O construtor desta classe tem um argumento representando a entrada a ser analisada.

A classe gerada tem um método público, sem argumentos, que deve ser utilizado no restante da aplicação para obter o próximo símbolo léxico da entrada. O nome default deste método é `yylex`, mas pode-se especificar um outro nome utilizando a opção `%function`. O tipo de retorno deste método de análise léxica é, por default, `YYToken`, que deve ser definido na aplicação. A opção `%type` permite usar um outro tipo, no entanto. Existe também a opção `%integer` que específica o tipo de retorno `int`.

Quando o final de arquivo é atingido, o método de análise léxica retorna um valor específico para indicar o fim da entrada. Quando o tipo de retorno for uma subclasse de `java.lang.Object`, o valor retornado é `null`, por default. Com `%integer` o valor de retorno default é a constante `YYEOF`, definida como uma variável de instância `public static final int` na classe gerada. Existem diferentes maneiras de se especificar um valor de retorno diferente do default quando se atinge o fim de arquivo.

A listagem 2 define a classe `Test` com o método `main` que pode ser utilizado para testar o analisador gerado.

Listagem 2: Classe para testar o analisador gerado pelo JFlex.

```java
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Reader;

public class Test
{
  public static void main(String[] args)
  {
    try
    {
      Reader input = args.length > 0 ?
                     new FileReader(args[0]) :
                     new InputStreamReader(System.in);
      Yylex scanner = new Yylex(input);
      int token;
      do
      {
        token = scanner.yylex();
        System.out.println(token);
      }
      while (token != Yylex.YYEOF);
    }
    catch (IOException e)
    {
      System.err.println(e);
    }
  }
}
```

Basicamente toma-se as seguintes ações:

1. O arquivo especificado na linha de comando é aberto para leitura
2. Uma instância da classe de análise léxica é criada, usando este arquivo como fonte
3. Em um comando de repetição, cada símbolo léxico é obtido e exibido na saída padrão, até que se chegue no final da entrada.

Tarefa 2

Cria um aplicativo Java para testar o JFlex utilizando a especificação e a classe `Test`, dadas anteriormente. Crie um programa de entrada e teste a aplicação determinando a saída para esta entrada.

5.4 Exemplo: usando uma classe para representar os símbols léxicos

A classe `Token`, definida na listagem 3, será utilizada para representar os símbolos léxicos.

Listagem 3: Classe para representar um símbolo léxico.

```java
import java.util.Formatter;

public class Token
{
  public enum T { IF, ID, INT, FLOAT, STR, EOF };

  public T type;
  public Object val;
  public int line;
  public int col;

  public Token(T type, int line, int col)
  {
    this.type = type;
    this.line = line;
    this.col = col;
  }

  public Token(T type, Object val, int line, int col)
  {
    this.type = type;
    this.val = val;
    this.line = line;
    this.col = col;
  }

  public String toString()
  {
    Formatter out = new Formatter();
    out.format("(%4d,%4d) %s", line, col, type);
    if (val != null)
      out.format(" [%s]", val);
    return out.toString();
  }
}
```

Cada símbolo léxico tem as seguintes informações:

- o tipo do símbolo léxico, representando a sua categoria, como por exemplo identificador, literal inteiro, literal em ponto flutuante, etc.
- valor semântico, que é alguma informação adicional sobre o símbolo léxico e que será necessária em etapas posteriores da compilação; exemplo: nome do identificador, valor de um literal inteiro, etc.
- posição, dada pelo número da linha e da coluna, em que o símbolo aparece na entrada.

Os tipos possíveis para um símbolo léxico são representados pelos valores enumerados do tipo `Token.T`.

A listagem 4 mostra uma especificação léxica simples.

Listagem 4: Exemplo de especificação léxica.

```
%%

%class Lexer
%type Token
%line
%column

%{
   private Token token(Token.T type)
   {
     return new Token(type, yyline, yycolumn);
   }

   private Token token(Token.T type, Object val)
   {
     return new Token(type, val, yyline, yycolumn);
   }
%}

%%

if             { return token(Token.T.IF); }
[a-z][a-z0-9]* { return token(Token.T.ID, yytext()); }
[0-9]+         { return token(Token.T.INT, new Integer(yytext())); }
[0-9]+"."[0-9]*|[0-9]*"."[0-9]+
               { return token(Token.T.FLOAT, new Double(yytext())); }
[ \t\n\r]+     { /* do nothing */ }
<<EOF>>        { return token(Token.T.EOF); }
.              { System.out.printf("error: unexpected char |%s|\n",
                                   yytext());
               }
```

Observe que:

- O nome da classe gerada é `Lexer`.
- O tipo do resultado do método que faz a análise léxica é `Token`.
- As opções `%line`, `%column` e `%char` habilitam a contagem de linhas, colunas e caracteres durante a análise léxica. As variáveis de instância `yyline`, `yycolumn` e `yychar` podem ser utilizadas para se obter a contagem de linha, de coluna e de caracter atual.
- As opções `%{` e `%}` delimitam um código que é inserido na classe gerado. Neste exemplo definimos dois métodos adicionais na classe gerada que facilitam a construção dos símbolos léxicos.
- A expressão regular `<<EOF>>` é utilizada quando o fim da entrada é atingido.

Tarefa 3

Explique porque a regra da palavra-chave `if` precisa ser colocada antes da regra de identificadores.

Tarefa 4

Utilize esta especificação léxica para gerar um analisador usando o JFlex. Crie uma nova classe `Test2` baseada na classe `Test` para testar o novo analisador. Teste o analisador.

5.5 Exemplo: definição de expressão regular

A listagem 5 mostra uma especificação léxica simples.

Listagem 5: Exemplo de especificação léxica.

```
%%

%class Lexer
%type Token
%line
%column

%{
    private Token token(Token.T type)
    {
      return new Token(type, yyline, yycolumn);
    }

    private Token token(Token.T type, Object val)
    {
      return new Token(type, val, yyline, yycolumn);
    }
%}

alpha = [a-zA-Z]
dig   = [0-9]
id    = {alpha} ({alpha} | {dig})*
int   = {dig}+
float = {dig}+ "." {dig}* | {dig}* "." {dig}+

%%

if         { return token(Token.T.IF); }
{id}       { return token(Token.T.ID, yytext()); }
{int}      { return token(Token.T.INT, new Integer(yytext())); }
{float}    { return token(Token.T.FLOAT, new Double(yytext())); }
[ \t\n\r]+ { /* do nothing */ }
<<EOF>>    { return token(Token.T.EOF); }
.          { System.out.printf("error: unexpected char |%s|\n",
                               yytext());
           }
```

Observe que segunda seção da especificação define cinco expressões regulares, `{alpha}`, `{dig}`, `{id}`, `{int}` e `{float}`, que são utilizadas nas próprias definições e nas regras léxicas.

Tarefa 5

Gere um analisador léxico usando esta nova especificação, e teste-o.

5.6 Exemplo: estados

A listagem 6 mostra uma especificação léxica que define novos estados.

Listagem 6: Exemplo de especificação léxica usando estados.

```
%%

%class Lexer
%type Token
%line
%column

%{
    private StringBuilder str = new StringBuilder();

    private Token token(Token.T type)
    {
      return new Token(type, yyline, yycolumn);
    }

    private Token token(Token.T type, Object val)
    {
      return new Token(type, val, yyline, yycolumn);
    }
%}

%state STR

alpha = [a-zA-Z]
dig   = [0-9]
id    = {alpha} ({alpha} | {dig})*
int   = {dig}+
float = {dig}+ "." {dig}* | {dig}* "." {dig}+

%%

<YYINITIAL> {
if                { return token(Token.T.IF); }
{id}              { return token(Token.T.ID, yytext()); }
{int}             { return token(Token.T.INT, new Integer(yytext())); }
{float}           { return token(Token.T.FLOAT, new Double(yytext())); }
\"                { str.setLength(0);
                    yybegin(STR);
                  }
[ \t\n\r]+        { /* do nothing */ }
<<EOF>>           { return token(Token.T.EOF); }
}

<STR> \"          { yybegin(YYINITIAL);
                    return token(Token.T.STR, str.toString());
                  }
<STR> \\t         { str.append('\t'); }
<STR> \\n         { str.append('\n'); }
<STR> \\\"        { str.append('"'); }
<STR> \\\\        { str.append('\\'); }
<STR> [^\n\r\\]+ { str.append(yytext()); }
<STR> <<EOF>>     { yybegin(YYINITIAL);
                    System.out.println("error: unclosed string literal");
                  }

.|\n              { System.out.printf("error: unexpected char |%s|\n",
                                      yytext());
                  }
```

Este analisador utiliza um estado especial para analisar literais caracteres. Observe que:

1. O estado inicial é `YYINITIAL`.
2. Outros estados podem ser declarados na segunda seção usando a opção `%state`.
3. O método `yybegin`, incluído pelo JFlex na classe gerada, permite mudar de estado.
4. Uma regra léxica pode ser prefixada com uma lista de estados, indicando que a regra só é usada se o analisador estiver em um dos etados listados.
5. Quanda a lista de estados é omitida de uma regra, a regra pode ser utilizada em qualquer estado.

Tarefa 6

Gere um analisador léxico usando esta nova especificação, e teste-o.