**Compiladores**
**Roteiro de Laboratório 01 – Construindo um Analisador Léxico**

# Parte I
# Utilizando o *lexer* do ANTLR

## 1 Introdução

No módulo de introdução da disciplina vimos que um compilador pode ser dividido em várias partes. A primeira destas partes é o analisador léxico, também chamado de *scanner* ou *lexer*. Este último termo é aquele comumente adotado pelo ANTLR, a ferramenta que iremos utilizar neste laboratório para a construção do analisador léxico.

O ANTLR é uma ferramenta para reconhecimento de linguagens, cujo desenvolvimento já ocorre há mais de 25 anos por Terence Parr. Em sua nova versão 4, o desenvolvimento do ANTLR teve ajuda do co-autor Sam Harwell.

O ANTLR foi escrito em Java, e assim, a ferramenta gera *lexers* escritos nesta mesma linguagem. No entanto, a partir da versão 4, o ANTLR passou a ser um gerador *multi-target*, isto é, agora a ferramenta consegue gerar saídas em várias linguagens diferentes além de Java, como Go, C++, C#, Python, JavaScript, Swift, dentre outras. Nestes roteiros de laboratório vamos nos ater à saída original em Java, por esta ser a mais madura e não apresentar *bugs* e outros problemas.

Se quiser mais informações sobre o ANTLR veja o site oficial em `https://www.antlr.org/about.html`.

## 2 Preparando o ambiente

### 2.1 Java e JVM

Para começar, é necessário ter instalado na máquina o compilador Java e a JVM (*Java Virtual Machine*) em alguma versão mais recente. Praticamente todas as distribuições Linux já incluem algum pacote do JDK (*Java Development Kit*) nos seus repositórios. Certifique-se que algum destes pacotes está instalado, e teste os comandos `javac` e `java` para garantir que tanto o compilador quanto a JVM estão acessíveis a partir do terminal.

### 2.2 Baixando o ANTLR

A seguir, devemos fazer o *download* do arquivo JAR do ANTLR no site da ferramenta. O ANTLR está em constante atualização, assim a sua versão mais recente muda regularmente. No momento da escrita deste roteiro, a versão mais atual é a 4.13.2 mas é possível que quando você estiver realizando esta atividade, a versão já tenha mudado. Veja o site `https://www.antlr.org/download` para determinar a versão mais recente e faça o download do arquivo JAR que inclui a versão completa dos binários do ANTLR.

O *link* direto de *download* para a versão 4.13.2 é `https://www.antlr.org/download/antlr-4.13.2-complete.jar`.

## 2.3 Diretório de instalação

Agora temos que escolher um diretório para colocar o arquivo JAR. Segundo a documentação do ANTLR, o local padrão é `/usr/local/lib`, mas essa não é uma boa opção porque você nem sempre vai ter acesso a esse diretório com a sua conta. Eu prefiro colocar o JAR dentro de um diretório `tools` e fazer um acesso relativo segundo um diretório raiz, por exemplo `labs`. Assim, o caminho de acesso ficaria algo como `./labs/tools/antlr-4.13.2-complete.jar`.

## 2.4 Configurando variáveis de ambiente

O guia de instalação rápida do ANTLR sugere a criação dos seguintes comandos. (Note que os comandos abaixo assumem que o ANTLR está no diretório padrão, vamos consertar isso depois.)

```
$ export CLASSPATH=".:/usr/local/lib/antlr-4.13.2-complete.jar:$CLASSPATH"
$ alias antlr4='java -jar /usr/local/lib/antlr-4.13.2-complete.jar'
$ alias grun='java org.antlr.v4.gui.TestRig'
```

A variável de ambiente `CLASSPATH` é acessada pelo compilador Java; ela deve estar devidamente configurada para que a compilação dos arquivos gerados pelo ANTLR dê certo. Já o comando `antlr4` serve para gerar o *lexer* a partir de uma especificação (mais detalhes a seguir). Por fim, o comando `grun` serve para executar o *lexer* gerado e realizar testes.

Como a variável `CLASSPATH` e os demais comandos são necessários o tempo todo, o tutorial do ANTLR sugere que estes comandos fiquem dentro de um arquivo de inicialização do terminal, como por exemplo o `bashrc`, para sistemas Linux. No entanto, nós vamos trabalhar de uma forma um pouco diferente, configurando todo o ambiente através de *Makefiles*. Isto será explicado adiante.

## 2.5 Executando o ANTLR

Para o ANTLR gerar o *lexer* ele precisa da definição dos *tokens* através de expressões regulares (ERs). Essas expressões ficam em um arquivo com a extensão `.g` ou `.g4`. Tendo este arquivo em mãos, podemos executar o ANTLR:

```
$ antlr4 -o dir_saida gramatica.g
```

Lembrando que o comando `antlr4` acima é só um *alias* para o comando real que executa o arquivo JAR: `java -jar antlr-4.13.2-complete.jar`. Assim, o comando anterior é equivalente a:

```
$ java -jar antlr-4.13.2-complete.jar -o dir_saida gramatica.g
```

Quando o ANTLR é executado ele gera 3 arquivos de saída: um `.java` que contém a implementação do *lexer*; e outros arquivos com as extensões `.token` e `.interop`, que contém algumas informações da representação interna dos *tokens* que o ANTLR usa. Geralmente estes podem ser ignorados.

O próximo passo da execução é compilar os arquivos `.java` criados:

```
$ javac ./dir_saida/*.java
```

(Note que para essa compilação dar certo a variável `CLASSPATH` precisa estar devidamente configurada. Uma outra alternativa é utilizar a opção `-cp` e informar o `CLASSPATH` diretamente

quando o `javac` é chamado. Essa é a forma que vamos usar nos exemplos. Veja a seção sobre *Makefiles* adiante.)

Agora que já temos o *lexer* em *bytecode*, podemos executá-lo para que ele reconheça os *tokens* da linguagem de entrada. Para isso, vamos usar o executor padrão do ANTLR, o `grun`:

```
$ grun nome_da_gramatica tokens -tokens
```

O argumento `tokens` é um nome artificial para a regra inicial da gramática. No caso temos somente a implementação do *lexer* então precisamos usar essa "regra". A opção `-tokens` indica que a gente só quer rodar o *lexer* (não há um *parser*), e os *tokens* reconhecidos devem ser exibidos na saída padrão. Ao executar um comando como acima, o terminal ficará aberto para a entrada dos *tokens*. Para finalizar é preciso pressionar `Ctrl+D` (no caso do Linux), ou `Ctrl+Z` (no caso do Windows).

Também é possível que o ANTLR realize a análise léxica de um arquivo de entrada (em vez de ler de `stdout`). Neste caso, basta colocar o caminho do arquivo no fim do comando, como por exemplo:

```
$ grun nome_da_gramatica tokens -tokens ./caminho/arquivo_de_entrada
```

Depois de finalizada a execução, você verá no terminal que o *lexer* retorna uma saída padrão tanto para quando ele reconhece os *tokens* como quando ele não reconhece. Nas próximas seções vamos dar vários exemplos de uso do ANTLR para tornar o entendimento da sua execução mais concreto.

## 2.6 Mantendo a sanidade com um `Makefile`

O método de configuração usual proposto no *site* do ANTLR funciona para casos simples, mas rapidamente fica repetitivo e chato para casos minimamente mais elaborados. Por conta disto, vamos usar sempre *Makefiles* para realizar a configuração do ANTLR.

Baixe o arquivo `CC_Lab01_Exemplos_Java.zip` do Classroom e veja os *Makefiles* dos exemplos. Todos têm o mesmo formato; o código abaixo mostra o `Makefile` do primeiro exemplo.

```
# Comando do compilador Java
JAVAC=javac
# Comando da JVM
JAVA=java
# ROOT é a raiz dos diretórios com todos os roteiros de laboratórios
YEAR=$(shell pwd | grep -o '20..-.')
ROOT=/home/zambon/Teaching/$(YEAR)/CC/labs
# Caminho para o JAR do ANTLR em labs/tools
ANTLR_PATH=$(ROOT)/tools/antlr-4.13.2-complete.jar
# Opção de configuração do CLASSPATH para o ambiente Java
CLASS_PATH_OPTION=-cp .:$(ANTLR_PATH)
# Configuração do comando de compilação do ANTLR
ANTLR4=$(JAVA) -jar $(ANTLR_PATH)
# Configuração do ambiente de teste do ANTLR
GRUN=$(JAVA) $(CLASS_PATH_OPTION) org.antlr.v4.gui.TestRig
# Nome da gramática
GRAMMAR_NAME=Exemplo01
# Diretório para aonde vão os arquivos gerados
GEN_PATH=lexer
```

A primeira parte do `Makefile` listada contém uma série de variáveis de configuração, sendo que os comentários e as explicações anteriores devem ser suficientes para o entendimento de cada variável. Você pode utilizar exatamente esse mesmo formato para todos os seus códigos. Basicamente a única variável que precisa ser alterada é o nome da gramática.

Seguimos com a segunda parte do `Makefile`, que contém os comandos:

```
# Executa o ANTLR e o compilador Java
all: antlr javac
	@echo "Done."
# Executa o ANTLR para compilar a gramática
antlr: $(GRAMMAR_NAME).g
	$(ANTLR4) -o $(GEN_PATH) $(GRAMMAR_NAME).g
# Executa o javac para compilar os arquivos gerados
javac:
	$(JAVAC) $(CLASS_PATH_OPTION) $(GEN_PATH)/*.java
# Executa o lexer. Comando: $ make run FILE=arquivo_de_teste
run:
	cd $(GEN_PATH) && $(GRUN) $(GRAMMAR_NAME) tokens -tokens $(FILE)
# Remove os arquivos gerados pelo ANTLR
clean:
	@rm -rf $(GEN_PATH)
```

Os alvos acima deveriam ser auto-explicativos. Basicamente você só precisa chamar `make`, para executar o ANTLR e compilar os arquivos Java. Isto cria um diretório `lexer` para os arquivos gerados pelo ANTLR. Não é obrigatório fazer assim mas eu acho uma boa para evitar que os arquivos gerados pelo ANTLR fiquem misturados com o resto do código do compilador.

A seguir, basta fazer `make run` para executar o *lexer* com a entrada por `stdin`, ou `make run FILE=arquivo_de_entrada` para se ler de um arquivo.

# 3 Criando uma gramática para o ANTLR

## 3.1 Arquivo de entrada

O arquivo de entrada para o ANTLR é o arquivo que define as regras da gramática da linguagem analisada. O ANTLR permite a construção tanto de analisadores sintáticos quanto léxicos. Isto quer dizer que o ANTLR espera tanto as regras do *parser* quanto do *lexer* no arquivo da gramática. Como por agora nós só vamos construir o *lexer*, temos que informar isto no começo do arquivo. Um formato geral do arquivo `.g` que vamos usar aqui fica assim:

```
1 lexer grammar NOME_DA_GRAMATICA;
2
3 TIPO_DO_TOKEN  :  EXPRESSAO_REGULAR  { ACAO } ;
4 ...
```

É essencial destacar que, assim como em Java, o arquivo `.g` precisa ter o mesmo nome da gramática. Por exemplo, se a primeira linha de código estiver como `lexer grammar Teste`, então o arquivo deve se chamar `Teste.g`. Note que a capitalização faz diferença!

As demais linhas do arquivo começam com a definição do tipo do *token* que deve ser associado à ER que segue. É possível definir ações (código Java) que são executadas quando um ER é casada, embora no ANTLR isso não seja um comportamento recomendado. (Isto será justificado nos próximos laboratórios.)

## 3.2 Expressões regulares no ANTLR

Todos os *tokens* da linguagem de entrada precisam ser definidos através de expressões regulares (ERs) e nessa seção vamos mostrar como construí-las.

- Os meta-símbolos utilizados para especificação das expressões regulares são os abaixo.

  ```
  " ' \ [ ] ? - . * + | ( ) / { } % < > ~
  ```

  Se você quiser que esses símbolos representem os seus respectivos caracteres, você deve escapá-los com \ ou colocá-los entre aspas simples. Na dúvida, coloque toda a *string* que você quer reconhecer entre aspas, como por exemplo, `'xyz++'`.
- Os padrões para entrada do ANTLR utilizam um conjunto de ERs estendidas. Os comandos para construção dos padrões são:
  - `'xyz'`: reconhece a sequência de caracteres `xyz`.
  - `.`: reconhece qualquer caractere **inclusive enter**.
  - `[xyz]`: uma *classe de caracteres*, nesse caso, a ER casa com o caractere `x`, **ou** o caractere `y` **ou** `z`. Isto é equivalente à ER `(x|y|z)`.
  - `[abj-oZ]`: uma classe de caracteres contendo uma *faixa*. Casa com `a`, ou `b`, ou qualquer letra de `j` a `o`, ou por fim `Z`.
  - `[a-zA-Z]`: reconhece uma letra do alfabeto (maiúscula ou minúscula).
  - `[ \t\n]` reconhece um espaço em branco ou um *tab* ou uma quebra de linha.
  - `('a'..'z')`: uma forma alternativa de declarar intervalos, reconhece qualquer caractere no intervalo de **a** a **z**.
  - `~[A-Z]`: uma classe de caracteres *negada*, isto é, qualquer caractere que não esteja na classe. Nesse exemplo, casa com qualquer caractere exceto letras maiúsculas.
  - `r*`: zero ou mais ocorrências da ER `r`.
  - `r+`: uma ou mais ocorrências de `r`.
  - `r?`: zero ou uma ocorrência de `r` (isto é, um `r` opcional).
  - `(r)`: parênteses são usados para agrupar sub-expressões.
  - `r|s`: reconhece `r` ou `s`.
- *Obs.:* Note que, dentro de uma classe de caracteres, todos os meta-símbolos do ANTLR são tratados como caracteres normais, com exceção dos caracteres \ e -.

Caso uma ER possa ser utilizada como parte da definição de outras ERs, temos a opção `fragment`. Com ela você define a sua ER de base e depois pode chamá-la na criação das outras, como neste exemplo:

```
1 fragment DIGITO : [0-9];
2 DEZENA  : DIGITO DIGITO;
3 CENTENA : DIGITO DIGITO DIGITO;
```

## 3.3 Exemplos básicos de *lexers* no ANTLR

### 3.3.1 Exemplo 01 - *Hello, World!*

Vamos começar pelo exemplo obrigatório do *Hello World* em ANTLR. Queremos uma gramática que reconheça as palavas *Hello* e *World*, além dos elementos de pontuação `,` e `!`. O código abaixo mostra o conteúdo do arquivo `Exemplo01.g`, aonde a primeira regra (`WS` – *whitespace*) serve para ignorar quaisquer espaços, quebra de linha e tabulação.

```
1 lexer grammar Exemplo01;
```

```
2
3 WS     : [ \t\n]+ -> skip ;
4 HELLO  : 'Hello' ;
5 WORLD  : 'World' ;
6 COMMA  : ',' ;
7 EXCLAM : '!' ;
```

Faça o *download* do arquivo `CC_Lab01_Exemplos_Java.zip` e a seguir abra um terminal no diretório `ex01`. Para compilar e executar o *lexer*, basta usar os comandos do *Makefile*, como explicado anteriormente. Lembre de fechar o *stream* do `stdin` com `Ctrl+D` após digitar a entrada.

```
$ make
$ make run
Hello, World!
[@0,0:4='Hello',<'Hello'>,1:0]
[@1,5:5=',',<',',1:5]
[@2,7:11='World',<'World'>,1:7]
[@3,12:12='!',<'!'>,1:12]
[@4,14:13='<EOF>',<EOF>,2:0]
```

O resultado da execução é a saída padrão do ANTLR com a opção `-tokens`, que exibe sequencialmente os *tokens* reconhecidos, um por linha. Vamos detalhar agora os campos de cada *token*. Por exemplo, a impressão `[@2,7:11='World',<'World'>,1:7]` nos diz que este *token* está no índice 2; que ele vai do caractere de entrada 7 até o 11; que o lexema reconhecido foi `'World'`; que o tipo do *token* é `<'World'>`; e que o *token* ocorreu na linha 1, posição 7. Quando a relação entre tipo do *token* e lexema é de 1 para 1, o ANTLR utiliza o próprio lexema para indicar o tipo do *token* na saída. Já quando a relação é de 1 para muitos, o tipo do *token* é o nome indicado na gramática. (Veja a gramática do Exemplo 02, adiante.)

Na execução anterior, vimos que o ANTLR reconheceu toda a entrada corretamente. Porém se inserirmos na entrada uma sequência de caracteres que não está definida na gramática, teremos uma indicação de erros:

```
$ make run
Hello, Crazy World!
line 1:7 token recognition error at: 'C'
line 1:8 token recognition error at: 'r'
line 1:9 token recognition error at: 'a'
line 1:10 token recognition error at: 'z'
line 1:11 token recognition error at: 'y'
[@0,0:4='Hello',<'Hello'>,1:0]
[@1,5:5=',',<',',1:5]
[@2,13:17='World',<'World'>,1:13]
[@3,18:18='!',<'!'>,1:18]
[@4,20:19='<EOF>',<EOF>,2:0]
```

### 3.3.2 Exemplo 02 - Reconhecendo operações aritméticas, números reais e inteiros

Neste próximo exemplo, queremos analisar as quatro operações aritméticas básicas, permitindo o uso de números inteiros e de ponto flutuante, com ou sem sinal. Isto pode ser feito com a

gramática seguinte.

```
lexer grammar Exemplo02;

fragment DIGITS : [0-9]+ ;

WS : [ \t\n]+ -> skip ;

PLUS  : '+' ;
MINUS : '-' ;
TIMES : '*' ;
OVER  : '/' ;

POS_INT : DIGITS ;
NEG_INT : '-' DIGITS;
POS_REAL : DIGITS '.' DIGITS ;
NEG_REAL : '-' DIGITS '.' DIGITS ;
```

Note o uso do fragmento `DIGITS` no exemplo acima para evitar repetição de uma mesma expressão regular. Isto pode deixar o código da gramática um pouco mais fácil de ler em alguns casos. Compilando e executando para a entrada `3 + 5 - -10 * 5.25 / -2.1`:

```
$ make
$ make run <<< "3 + 5 - -10 * 5.25 / -2.1"
[@0,0:0='3',<POS_INT>,1:0]
[@1,2:2='+',<'+'>,1:2]
[@2,4:4='5',<POS_INT>,1:4]
[@3,6:6='-',<'-'>,1:6]
[@4,8:10='-10',<NEG_INT>,1:8]
[@5,12:12='*',<'*'>,1:12]
[@6,14:17='5.25',<POS_REAL>,1:14]
[@7,19:19='/',<'/'>,1:19]
[@8,21:24='-2.1',<NEG_REAL>,1:21]
[@9,26:25='<EOF>',<EOF>,2:0]
```

### 3.3.3 Exemplo 03 - Reconhecendo algumas palavras chaves, variáveis e *strings*

Nesse exemplo vamos reconhecer alguns dos *tokens* básicos de uma linguagem de programação.

```
lexer grammar Exemplo03;

WS : [ \t\n]+ -> skip ;

IF   : 'if' ;
ELSE : 'else' ;
TRUE : 'true' ;

ASSIGN : '=' ;
STRING : '"' ~["]* '"' ;
ID     : [a-zA-Z]+ ;
```

Se executarmos o *lexer* para a entrada abaixo:

```
if true
    x = "then"
else
    x = "false"
```

A saída será:

```
[@0,0:1='if',<'if'>,1:0]
[@1,3:6='true',<'true'>,1:3]
[@2,12:12='x',<ID>,2:4]
[@3,14:14='=',<'='>,2:6]
[@4,16:21='"then"',<STRING>,2:8]
[@5,23:26='else',<'else'>,3:0]
[@6,32:32='x',<ID>,4:4]
[@7,34:34='=',<'='>,4:6]
[@8,36:42='"false"',<STRING>,4:8]
[@9,44:43='<EOF>',<EOF>,5:0]
```

Note que podem ocorrer conflitos de reconhecimento entre as palavras chaves e IDs. Como esperado, o ANTLR trata isso pela ordem em que os *tokens* são definidos. Experimente colocar o ID antes das palavras chaves e veja como ficará o reconhecimento dos *tokens*.

### 3.4 Exercícios de Aquecimento

0. Faça o download dos arquivos de exemplo. Compile-os e execute-os como explicado acima. Observe os arquivos gerados pelo ANTLR, abra-os e veja se reconhece funções e como ele organiza as estruturas de dados. Tente executar os exemplos com arquivos de entrada ao invés de `stdin` como feito nos testes.
1. Remova de um arquivo de entrada todas as ocorrências de `#` e o restante da linha. Isto é útil para eliminar comentários em *scripts shell*, por exemplo. (*Obs. 1*: Será necessário o uso de ações (ditas "semânticas") neste exercício. *Obs. 2*: O método `getText()` pode ser usado em uma ação para obter o lexema identificado.)
2. Encontre letras maiúsculas na entrada e substitua pelas suas equivalentes em minúsculas. Não modifique os demais caracteres.
3. Reconheça inteiros 32-bit em notação hexadecimal. Os números começam com `0x` ou `0X` e podem ter no máximo 8 dígitos hexadecimais. As letras podem ser em qualquer caixa (alta ou baixa).
4. Reconheça placas de carros antigas no formato `AAA-0000`.

# Parte II
# Construindo um *lexer* para EZLang

## 4 A Linguagem EZLang

Um programa em EZLang tem uma estrutura bastante simples: ele é composto apenas por uma sequência de declarações (*statements*) separadas por ponto-e-vírgula, com uma sintaxe similar a de Ada e Pascal. Não existem funções ou procedimentos, somente o corpo do programa principal. Existem apenas quatro tipos primitivos (inteiro, real, Booleano e *string*) e não é possível criar novos tipos. Existem apenas dois comandos de controle: `if` e `repeat`. Ambos podem conter um bloco de comandos. Um comando `if` pode ter uma parte `else` opcional e deve terminar com a palavra-chave `end`. Existem também comandos `read` e `write` para realização de operações de entrada e saída básicas. Comentários são escritos entre chaves e não podem ser aninhados.

Expressões em EZLang são limitadas a expressões sobre os tipos primitivos. Uma expressão Booleana consiste de uma comparação entre duas expressões aritméticas usando um dos dois operadores de comparação, < ou =. Uma expressão aritmética pode envolver constantes numéricas, variáveis, parênteses e quaisquer dos quatro operadores aritméticos +, −, ⋆ e /, com as propriedades matemáticas usuais.

Apesar da linguagem EZLang não possuir muitas das características necessárias a linguagens de programação reais (procedimentos, vetores e estruturas estão entre as omissões mais sérias), a linguagem ainda é suficientemente grande para exemplificar a maioria das características essenciais de um compilador.

Na listagem abaixo temos um exemplo de um programa para o cálculo da função fatorial.

```
1  { Sample program in EZ language -
2    computes factorial
3  }
4  program fact;
5  var
6      int x;
7      int fact;
8  begin
9      read x; { input an integer }
10     if 0 < x then { don't compute if x <= 0 }
11         fact := 1;
12         repeat
13             fact := fact * x;
14             x := x - 1;
15         until x = 0
16         write fact; { output factorial of x }
17     end
18 end
```

# 5 Convenções léxicas da linguagem EZLang

A tarefa deste laboratório é construir um *lexer* para EZLang. Eis os tipos de *tokens* e seus respectivos lexemas.

```
1 { Palavras reservadas: }
2 begin   bool   else    end     false   if      int     program
3 read    real   repeat  string  then    true    until   var     write
4
5 { Símbolos especiais: }
6 :=  =   <   +   -   *   /   (   )   ;
7
8 { Constantes numéricas e strings (exemplos): }
9 42      4.2     "abc"
```

Os *tokens* de EZLang podem ser classificados em três categorias típicas: palavras reservadas, símbolos especiais e outras. Há 17 palavras reservadas, com os significados usuais (embora a sua semântica só será propriamente definida em laboratórios futuros). Existem 10 símbolos especiais, para as quatro operações aritméticas básicas, duas operações de comparação (igual e menor que), parênteses, ponto-e-vírgula e atribuição. Todos os símbolos especiais têm comprimento de um caractere, exceto a atribuição, que tem dois caracteres. Os outros *tokens* são números, que são sequências com um ou mais dígitos, *strings*, e identificadores, que (para simplificar) são sequências com uma ou mais letras.

Além dos *tokens*, EZLang segue as convenções léxicas a seguir. Comentários são cercados por chaves e não podem ser aninhados. O formato do código é livre, isto é, não existem colunas ou posições específicas para uma dada operação. Espaços são formados por branco, tabulações e quebras de linhas.

# 6 Implementado um *lexer* para a linguagem EZLang

Você deve criar um *lexer* que reconhece todos os elementos léxicos da linguagem EZLang. Para tal, utilize o ANTLR. Como já visto anteriormente, o ANTLR implementa a sua própria saída padrão para os *tokens* reconhecidos e você pode continuar usando-a.

A saída do seu *lexer* para o programa do exemplo anterior deve ficar da seguinte forma:

```
[@0,57:63='program',<'program'>,4:0]
[@1,65:68='fact',<ID>,4:8]
[@2,69:69=';',<';'>,4:12]
[@3,71:73='var',<'var'>,5:0]
[@4,79:81='int',<'int'>,6:4]
[@5,83:83='x',<ID>,6:8]
[@6,84:84=';',<';'>,6:9]
[@7,90:92='int',<'int'>,7:4]
[@8,94:97='fact',<ID>,7:8]
[@9,98:98=';',<';'>,7:12]
[@10,100:104='begin',<'begin'>,8:0]
[@11,110:113='read',<'read'>,9:4]
[@12,115:115='x',<ID>,9:9]
[@13,116:116=';',<';'>,9:10]
```

```
[@14,143:144='if',<'if'>,10:4]
[@15,146:146='0',<INT_VAL>,10:7]
[@16,148:148='<',<'<'>,10:9]
[@17,150:150='x',<ID>,10:11]
[@18,152:155='then',<'then'>,10:13]
[@19,193:196='fact',<ID>,11:8]
[@20,198:199=':=',<':='>,11:13]
[@21,201:201='1',<INT_VAL>,11:16]
[@22,202:202=';',<';'>,11:17]
[@23,212:217='repeat',<'repeat'>,12:8]
[@24,231:234='fact',<ID>,13:12]
[@25,236:237=':=',<':='>,13:17]
[@26,239:242='fact',<ID>,13:20]
[@27,244:244='*',<'*'>,13:25]
[@28,246:246='x',<ID>,13:27]
[@29,247:247=';',<';'>,13:28]
[@30,261:261='x',<ID>,14:12]
[@31,263:264=':=',<':='>,14:14]
[@32,266:266='x',<ID>,14:17]
[@33,268:268='-',<'-'>,14:19]
[@34,270:270='1',<INT_VAL>,14:21]
[@35,271:271=';',<';'>,14:22]
[@36,281:285='until',<'until'>,15:8]
[@37,287:287='x',<ID>,15:14]
[@38,289:289='=',<'='>,15:16]
[@39,291:291='0',<INT_VAL>,15:18]
[@40,301:305='write',<'write'>,16:8]
[@41,307:310='fact',<ID>,16:14]
[@42,311:311=';',<';'>,16:18]
[@43,343:345='end',<'end'>,17:4]
[@44,347:349='end',<'end'>,18:0]
[@45,351:350='<EOF>',<EOF>,19:0]
```

Veja os exemplos no arquivo de entrada (`in.zip`) com as respectivas saídas (arquivo `out01_java.zip`). Para testar a saída do seu *lexer* contra a fornecida pelo professor, use um *script* como abaixo:

```
#!/bin/bash

ROOT=labs
ANTLR_PATH=$ROOT/tools/antlr-4.13.2-complete.jar
CLASS_PATH_OPTION="-cp .:$ANTLR_PATH"

GRAMMAR_NAME=EZLexer
GEN_PATH=lexer

DATA=$ROOT/io
IN=$DATA/in
OUT=$DATA/out01_java
```

```
cd $GEN_PATH
for infile in `ls $IN/*.ezl`; do
    base=$(basename $infile)
    outfile=$OUT/${base/.ezl/.out}
    echo Running $base
    java $CLASS_PATH_OPTION org.antlr.v4.gui.TestRig $GRAMMAR_NAME \
         tokens -tokens $infile 2>&1 | diff -w $outfile -
done
```

Uma implementação de referência para esse laboratório será disponibilizada pelo professor em um futuro próximo. No entanto, você é *fortemente* encorajado a realizar a sua implementação completa antes de ver uma solução em outro lugar.