Bloco 2

Introdução ao ANTLR

Resumo:

- Instalação e utilização do ANTLR.
- Construção de gramáticas simples.
- Introdução à programação em ANTLR.

Instalação do ANTLR4

Descarregue o ficheiro `antlr4-install.zip` disponível no elearning, extraia o seu conteúdo, abra um terminal no directório `antlr4-install` e, finalmente, execute o comando `./install.sh`.

Exercício 2.01

Defina a seguinte gramática no ficheiro `Hello.g4`[1]:

```
grammar Hello;              // Define a grammar called Hello
greetings : 'hello' ID ;    // match keyword hello followed by an identifier
ID : [a-z]+ ;               // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ;   // skip spaces, tabs, newlines, \r (Windows)
```

a) Experimente compilar a gramática `Hello.g4`, e executar o tradutor resultante (utilize o comando `antlr4-test` para testar a gramática).

b) Acrescente um *visitor* (`Execute`) de modo a, no fim da regra `greetings`, escrever em português: `Olá <ID>` (para gerar um *visitor* (`HelloBaseVisitor`) passe o argumento `-visitor` ao comando `antlr4`).

Note que o `ANTLR` por cada regra sintáctica `r: a b;`, cria um método `visitR` que tem como argumento o contexto dessa regra (`*Parser.RContext ctx`). Esse

[1] Não copie o código do ficheiro pdf, já que alguns caracteres copiados poderão não ser ASCII (gerando erros na compilação do antlr4).

contexto é definido pela classe `*Parser.RContext` que, caso `a` e `b` sejam não terminais, contém métodos que devolvem os respectivos contextos `*Parser.AContext` e `*Parser.BContext`. Caso se pretenda visitar um desses contextos, basta no método `visitR`, invocar, por exemplo, `visit(ctx.a())`.

c) Acrescente uma regra de despedida à gramática (`bye ID`), fazendo com que a gramática aceite uma qualquer das regras (`greetings`, ou `bye`). (Para definir em `ANTLR` uma regra com mais do que uma alternativa utiliza-se o separador `|`, por exemplo: `r: a | b ;`.)

d) Generalize os identificadores por forma a incluir letras maiúsculas e a permitir nomes com mais do que um identificador. (Em `ANTLR` uma regra com uma ou mais repetições e definida por `r+`, por exemplo: `r: a+ ;`.)

e) Generalize a gramática por forma a permitir a repetição até ao fim do ficheiro das regras de qualquer uma das regras atrás descritas (`greetings`, ou `bye`).

Exercício 2.02
Considere a seguinte gramática (`SuffixCalculator.g4`):

```
grammar SuffixCalculator;

program:
    stat* EOF      // Zero or more repetitions of stat
  ;

stat:
    expr? NEWLINE  // Optative expr
  ;

expr:
    expr expr op=('*'|'/'|'+'|'-')  #ExprSuffix
  | Number                          #ExprNumber
  ;

Number: [0-9]+('.'[0-9]+)?;
NEWLINE: '\r'? '\n';
WS: [ \t]+ -> skip;
```

a) Experimente compilar esta gramática e executar o tradutor resultante.

b) Utilizando esta gramática e acrescentando um *visitor* (`Interpreter`), tente implementar uma calculadora em notação pós-fixa (sufixa, ou *Reverse Polish Notation*) para as operações aritméticas elementares definidas (ou seja, que efetue os cálculos e apresente os resultados para cada linha processada).

Note que em `ANTLR` podemos associar *visits*/*callbacks* a diferentes alternativas numa regra sintáctica:

```
r : a #altA
  | b #altB
  | c #altC
  ;
```

Neste caso, irão ser criados *visits/callbacks* quer nos *visitors* quer nos *listeners*, para as três alternativas apresentadas, não aparecendo o *visit/callback* para a regra `r`.

Exercício 2.03
Considere a seguinte gramática para uma calculadora de números inteiros (`Calculator.g4`):

```
grammar Calculator;

program:
      stat* EOF
    ;

stat:
      expr? NEWLINE
    ;

expr:
      expr op=('*'|'/'|'%') expr   #ExprMultDivMod
    | expr op=('+'|'-') expr       #ExprAddSub
    | Integer                      #ExprInteger
    | '(' expr ')'                 #ExprParent
    ;

Integer: [0-9]+;  // implement with long integers
NEWLINE: '\r'? '\n';
WS: [ \t]+ -> skip;
COMMENT: '#' .*? '\n' -> skip;
```

a) Experimente compilar esta gramática e executar o tradutor resultante.

b) Utilizando esta gramática e acrescentando um *visitor* (`Interpreter`), tente implementar uma calculadora para as operações aritméticas elementares definidas (ou seja, que efetue os cálculos e apresente os resultados para cada linha processada).

c) Acrescente os operadores unários + e −.

   Note que a ambiguidade entre operadores binários e unários que utilizam o mesmo símbolo (+ e −) só pode ser resolvida sintacticamente, e que estes operadores devem ter precedência sobre todos os outros (por exemplo, na expressão `-3+4` o operador unário − aplica-se ao `3` e não ao resultado da soma).

Exercício 2.04
Implemente uma gramática para fazer a análise sintáctica dos ficheiros utilizados no exercício 1.03. Utilizando um *listener*, altere a resolução desse problema por forma a incluir esta gramática na solução.

Exercício 2.05
Altere o problema 2.03 acrescentando a possibilidade de definir e utilizar variáveis. Para esse fim considere uma nova instrução (i.e. uma nova alternativa no `stat`):

```
stat: ...  // make necessary changes
assignment: ID '=' expr;
...
expr:
    ...
    | ID                          #ExprId
    ...
ID: [a-zA-Z_]+ ;
...
```

Para dar suporte ao registo dos valores associados a variáveis, utilize um *array* associativo (java.util.HashMap).

Exercício 2.06
Descarregue a gramática da linguagem `Java` (`https://github.com/antlr/grammars-v4`), e experimente fazer a análise sintáctica de programas `Java` simples (versão 8).

Utilize o suporte para *listeners* do `ANTLR` para escrever o nome da classe e dos métodos sujeitos a análise sintáctica.

Exercício 2.07
Utilizando a gramática definida no problema 2.05 e recorrendo aos *visitors* do `ANTLR`, converta uma expressão aritmética infixa (operador no meio dos operandos), numa expressão equivalente sufixa (operador no fim). Para resolver o problema de ambiguidade com os operadores unários + e −, faça com que na expressão sufixa convertida, esses operadores apareçam, respectivamente, como `!+` e `!-`. Por exemplo:

- $2 + 3 \quad \rightarrow \quad 2\ 3\ +$
- $a = 2 + 3 * 4 \quad \rightarrow \quad a\ =\ 2\ 3\ 4\ *\ +$
- $3 * (2 + 1) + (2 - 1) \quad \rightarrow \quad 3\ 2\ 1\ +\ *\ 2\ 1\ -\ +$
- $3 * +(4/2) + (-2 - 1) \quad \rightarrow \quad 3\ 4\ 2\ /\ !+\ *\ 2\ !-\ 1\ -\ +$

Exercício 2.08
Pretende-se implementar uma calculadora para fracções racionais (numerador e denominador representados por números inteiros).

Defina uma gramática para esta linguagem tendo em conta o seguinte programa exemplo (os operadores aritméticos devem ter as precedências habituais):

```
// basic:
print 1/4;   // escreve na consola a fracção 1/4
print 3;     // escreve na consola a fracção 3
3/4 -> x;    // guarda a fracção 3/4 na variável x
print x;     // escreve na consola a fracção armazenada na variável x
// more advanced:
print 1/4-(1/4);
```

```
4/2 * (-2/3 + 4) - 2 -> x;
print x;
print x*x:x+(x)-x;    // a divisão de fracções é feita pelo operador :
print (1/2)^3;        // operador potência para expoentes inteiros (base entre parêntesis)
print reduce 2/4;     // redução de fracções
```

Nota 1: Tente tirar o melhor proveito possível das instruções exemplificadas por forma a tornar a linguagem o mais genérica possível. No entanto, pode considerar as fracções literais (ex: $1/4$, 2, $-3/2$) são sempre ou um número inteiro (i.e. denominador unitário), ou uma razão entre dois inteiros literais (em que só o numerador pode ser negativo).

Nota 2: Existem ficheiros `*.txt` que exemplificam diversos programas.

Implemente este interpretador com *visitors*.

Exercício 2.09

Crie um programa, com base em código de reconhecimento de estruturas de dados gerado pelo `ANTLR4`, que permita não só gerir uma base de dados de perguntas de escolha múltipla, como também aplicar essa base de dados para gerar uma pergunta com uma escolha múltipla aleatória. Este programa deverá receber como entrada o nome de um ficheiro com a base de dados das perguntas possíveis, um texto a identificar a pergunta a gerar, e o número de alíneas a gerar.

A estrutura das perguntas presente no ficheiro de base de dados deverá ser a seguinte (comentários e espaços em branco devem ser ignorados):

```
# comentário de linha
identificação-pergunta(texto-da-pergunta) {
  texto-de-uma-resposta:cotação;
  texto-de-uma-resposta:cotação;
  ...
}
...
```

Na identificação da pergunta pode-se aceitar letras, números e pontos. Um grupo de perguntas é referenciado por um padrão que tem de ser um sub-texto de identificadores de perguntas (pode referenciar mais do que uma pergunta). Como é natural que as perguntas e respostas tenham mais do que uma palavra, facilita a resolução do problema considerar a delimitação das mesmas por símbolos especiais (por exemplo, aspas).

Exemplo de um ficheiro possível (`bd-1.question`):

```
# ANTLR:
antlr.P1("O ANTLR4 permite...") {
  "construir um interpretador para uma linguagem":100;
  "construir um compilador para uma linguagem":100;
  "paralelizar um programa":0;
  "processar ficheiros de especificação de linguagens concebidos para lex e yacc":0;
  "a geração automática de código de processamento de
   uma linguagem em qualquer outra linguagem":0;
  "verificar se um programa está correcto":0;
}
```

```
# Outras áreas:
outro.P1("Um gato é...") {
  "um insecto":0;
  "um mamífero":100;
  "o melhor amigo do Homem":0;
  "um animal da família dos felinos":100;
  "um peixe":0;
}
```

A seguinte invocação do programa – `b2_8 bd-1.question P1 3`, – poderia gerar a seguinte saída:

```
- Um gato é...
   a) um peixe;
   b) o melhor amigo do Homem;
   c) um mamífero.
```

Exercício 2.10
Pretende-se desenvolver uma calculadora simples para operações sobre conjuntos. Nesta versão simplificada, vamos restringir os conjuntos a listas finitas de elementos, definidos por extensão.

Desenvolva uma gramática para esta calculadora, tendo em consideração a seguinte especificação:

- Um conjunto é definido por uma sequencia de palavras, ou de números, separada por vírgulas e delimitada por chavetas:

  $\{a, b, c\}$, $\{1, 3, 5, 7, 9\}$

- Uma palavra é uma sequência de letras minúsculas.

- Um número é uma sequência de dígitos, eventualmente precedida pelo sinal menos ou pelo sinal mais.

- Pode-se definir (ou redefinir) variáveis que representem conjuntos com uma instrução de atribuição de valor:

  $C = \{a, b, c\}$

- Uma variável é uma sequência de letras maiúsculas.

- Implemente as operações (com prioridade crescente) sobre conjuntos: união (definida pelo símbolo +), intercepção (símbolo &) e diferença (símbolo \).

  $\{a, b, c\} + \{b, d\}$

- Para poder definir diferentes precedências, implemente os parêntesis.

  $(\{a\} + \{b\}) \setminus \{b\}$

- Implemente comentários de linha definidos pelo prefixo `--`.

  ```
  -- this is a comment!
  ```

- Considere que a calculadora funciona como interpretador, em que cada linha representa uma instrução cujo resultado será um conjunto que deve ser apresentado:

  $\mathrm{C} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$

  result: $\{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$

  $\{\mathrm{a}, \mathrm{b}, \mathrm{c}\} \ + \ \{\mathrm{b}, \mathrm{d}\}$

  result: $\{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}$

  $\{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$ & $\{\mathrm{b}, \mathrm{d}\}$

  result: $\{\mathrm{b}\}$

  $\{\mathrm{a}, \mathrm{b}, \mathrm{c}\} \setminus \{\mathrm{b}, \mathrm{d}\}$

  result: $\{\mathrm{a}, \mathrm{c}\}$

  $\mathrm{C} \setminus \mathrm{C}$

  result: $\{\}$

Implemente interpretadores para esta linguagem com as seguintes variantes (separadas em pacotes `java` diferentes):

a) Com acções e atributos na gramática (*package*: `ActionsAndAttributes`);

b) Com atributos e *listeners* (*package*: `ListenersAndAttributes`);

c) Com o *array* associativo `ParseTreeProperty` e *listeners* (*package*: `ListenersAndPTP`);

d) Só com *visitors* (*package*: `Visitors`).