Linguagens Formais e Autómatos
Resolução dos exercícios do bloco 2

Tomás Oliveira e Silva (com o apoio de Miguel Oliveira e Silva)

27 de Abril de 2017 (revisto em Março de 2018 e Março de 2019)

Parte-se do pressuposto que o `Java` e o `ANTLR`, versão 4.7.x, estão corretamente instalados, e que os *scripts* de apoio à utilização do `ANTLR`, como por exemplo, `antlr4` e `antlr4-main`, também estão bem instalados. Recomenda-se vivamente a leitura dos primeiros capítulos do livro The Definitive ANTLR 4 Reference, escrito pelo autor do ANTLR (Terence Parr).

Exercício 2.01

Grave no ficheiro `Hello.g4` a seguinte gramática `ANTLR`:

```
grammar Hello ;              // o nome da gramática é Hello
greetings : 'hello' ID ;     // a regra de entrada da gramática
ID : [a-z]+ ;                // um ID é composto por uma ou mais letras minúsculas
WS : [ \t\r\n]+ -> skip ;    // espaços, tabs, e mudanças de linha são descartados
```

a) Experimente compilar a gramática `Hello.g4`, e executar o tradutor resultante.

Proposta de solução. Num terminal, execute os seguintes comandos (`'$'` é aqui o *prompt* da *shell* — no seu caso deve aparecer algum texto antes do cifrão — apenas deve introduzir o texto que aparece depois do primeiro cifrão):

```
$ antlr4 Hello.g4
$ javac *.java           # O comando antlr4-build executa estas duas compilações
$ echo "hello aluna" | antlr4-test Hello greetings
```

O primeiro comando gera o *parser* e o *lexer* para a gramática especificada no ficheiro `Hello.g4`. Isto é feito usando o `ANTLR`. O segundo comando compila todo o código `Java`. O terceiro comando corre o *TestRig* do `ANTLR` para a gramática `Hello` usando `greetings` como regra principal. Se tudo estiver em ordem, não é produzido nenhum texto. Podemos depois também executar

```
$ echo "hello aluna" | antlr4-test Hello greetings -tokens
$ echo "hello aluna" | antlr4-test Hello greetings -gui
```

para ver os *tokens* que foram gerados e para ver a árvore sintática que foi gerada. Também é possível fazer algo do género (desde o início):

```
$ antlr4-main Hello greetings
$ antlr4 Hello.g4
$ javac *.java
$ echo "hello aluna" | java -ea HelloMain    # Pode também usar o comando antlr4-run
```

O segundo comando gera código `Java` que define uma classe com o nome `HelloMain`. O `Hello` da parte inicial do nome dessa classe é o primeiro argumento do *script* `antlr4-main`, e tem de coincidir com o nome da gramática. O segundo argumento desse *script* é o nome da regra de entrada (a primeira) da gramática. O quarto executa o código compilado, fornecendo-lhe como texto de entrada o `hello aluna`; se tudo correr bem, não deve produzir qualquer saída.

b) Altere a gramática de modo a incluir uma ação no fim da regra `greetings` escrevendo em português: `Olá <ID>` (onde está `<ID>` deve aparecer o texto que de facto foi introduzido).

Proposta de solução. Altere o ficheiro `Hello.g4` de modo a ficar como se segue:

```
grammar Hello ;
greetings : 'hello' ID { System.out.println("Olá " + $ID.text); } ;
ID : [a-z]+ ;
WS : [ \t\n\r]+ -> skip ;
```

Na segunda linha da gramática introduzimos algum código `Java`, escrito dentro das chavetas, que é executado quando a regra é aceite. Nesse código podemos aceder a toda a informação sobre o *token* `ID` que foi aceite, que no nosso caso se resume ao seu texto (`.text`). Note que é preciso colocar um cifrão antes de `ID`, para que o `ANTLR` coloque nesse sítio o código `Java` apropriado para fazer referência à variável que contem toda a informação sobre o *token*.

Para experimentar a nova gramática execute, por exemplo:

```
$ antlr4 Hello.g4
$ javac *.java
$ echo "hello aluna" | java HelloMain
```

Agora a execução deverá produzir o texto `Olá aluna`.

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 uma regra com mais do que uma alternativa utiliza-se o separador `|`, por exemplo: "`regra : alternativa1 | alternativa2 ;`".)

Proposta de solução. Altere o ficheiro `Hello.g4` de modo a ficar como se segue:

```
grammar Hello;
top : greetings | bye ;
greetings : 'hello' ID { System.out.println("Olá " + $ID.text); } ;
bye       : 'bye'   ID { System.out.println("Adeus " + $ID.text); } ;
ID : [a-z]+ ;
WS : [ \t\n\r]+ -> skip ;
```

Note que mudámos o nome à regra de entrada, a que passámos a chamar `top`, que tem duas alternativas: regra `greetings` ou regra `bye`. Para experimentar a nova gramática execute, por exemplo:

```
$ antlr4 Hello.g4
$ rm HelloMain.java
$ antlr4-main Hello top
$ javac *.java
$ echo "hello aluna bye aluno" | java HelloMain
```

Como mudámos o nome da regra de entrada da gramática foi necessário recriar a classe `HelloMain` (apagando primeiro o ficheiro existente, porque `antlr4-main` recusa-se a alterar

um ficheiro já existente). Agora a execução deverá produzir o texto `Olá aluna` numa linha (como a regra de entrada apenas aceita ou um `greetings` ou um `bye`, o texto `bye aluno` é ignorado).

d) Generalize os identificadores por forma a que sejam aceites identificadores compostos por uma ou mais palavras, sendo que cada palavra tem de começar por uma letra (maiúscula ou minúscula), seguida por zero ou mais letras (maiúsculas ou minúsculas), números (de 0 a 9), e *underscores*. Uma regra com uma ou mais repetições é definida por `regra+`, e uma regra com zero ou mais repetições é definida por `regra*`.

Proposta de solução. Altere o ficheiro `Hello.g4` de modo a ficar como se segue:

```
grammar Hello;
top : greetings | bye ;
greetings : 'hello' names { System.out.println("Olá "   + $names.list); } ;
bye       : 'bye'   names { System.out.println("Adeus " + $names.list); } ;
names returns[String list=""] : ( ID { $list = $list + ($list.isEmpty() ? "" : ",")
                                           + $ID.text; }
                                )+;
ID : [A-Za-z][A-Za-z0-9_]* ;
WS : [ \t\n\r]+ -> skip ;
```

Como queremos aceitar uma ou mais palavras, as regras `greetings` e `bye` passam a invocar a regra `names` (em vez de `ID`), que por sua vez aceita um ou mais `ID`. Isto é feito desta maneira, e não diretamente, para podermos construir, e ter acesso, ao texto por nós concatenado dos vários `ID` aceites, sem ter de percurrer toda a árvore sintática gerada pela execução do nosso programa. A regra `names` faz isso mesmo: aceita um ou mais `ID` — `( ID )+`, e guarda numa string (a que chamámos `list`) os vários `ID` aceites. A contatenação é feita com o código `Java` seguinte: `$list = $list + ($list.isEmpty() ? "" : ",") + $ID.text`. Mais uma vez, no código `Java` temos de usar um cifrão antes de `names` e `ID` para que o `ANTLR` introduza nesses sítios referências aos objetos apropriados. Depois de compilar esta nova versão da gramática, a execução do comando

```
$ echo "hello avz_x arn_7" | java HelloMain
```

deve produzir `Olá avz_x,arn_7`.

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

Proposta de solução. Altere o ficheiro `Hello.g4` de mode a que a sua segunda linha fique como se segue:

```
top : ( greetings | bye )* EOF ;
```

Com esta linha, passa-se a aceitar zero ou mais `greetings` e `bye`, intercalados arbitrariamente. Também se especificou que o nosso texto de entrada termina quando se chega ao fim do texto do *standard input*(`EOF` significa *End Of File*). Depois de compilar esta nova versão da gramática, a execução do comando

```
$ echo "hello avz_x arn_7 bye MOS tos" | java HelloMain
```

deve produzir `Olá avz_x,arn_7` numa linha e `Adeus MOS,tos` numa outra linha.

Exercício 2.03

Grave no ficheiro `Calculator.g4` a seguinte gramática `ANTLR`:

```
grammar Calculator ;
program : stat* EOF ;
stat : expr NEWLINE
     | NEWLINE ;
expr : expr ( '*' | '/' ) expr   // a alternativa mais prioritára
     | expr ( '+' | '-' ) expr
     | INT
     | '(' expr ')'              // a alternativa menos prioritária
     ;
INT : [0-9]+ ;                   // um INT é composto por um ou mais algarismos
NEWLINE : '\r'? '\n' ;           // o '\r' é opcional
WS: [ \t]+ -> skip ;
```

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

Proposta de solução. Ver solução da alínea a) do exercício 2.01.

b) Utilizando esta gramática e acrescentando ações e atributos nas regras, implemente uma calculadora para as operações aritméticas elementares definidas (ou seja, que efetue os cálculos e apresente os resultados para cada linha processada).

Proposta de solução. Vamos proceder como na solução da alínea d) do exercício 2.01. Precisamos de introduzir algum código `Java` na regra `expr`, de modo a que seja devolvido um `double` com o resultado da avaliação da expressão (eventuais recursões são tratadas automaticamente!), e temos de introduzir código na regra `stat` para imprimir o valor de um `expr`. Vamos ter de dar nomes (de variáveis) a partes de alternativas da regra `expr`, porque vamos precisar de fazer referência a essas partes no código `Java`. Por exemplo, em alguns casos vai ser preciso identificar se se trata de uma multiplicação ou de uma divisão. Para isso vamos substituir `( '*' | '/' )` por `op=( '*' | '/' )`, o que nos permite, por inspeção de `$op`, identificar a operação a efetuar. Finalmente, o código `Java` a introduzir em cada alternativa da regra `expr` terá de fazer a operação apropriada para esse alternativa. O resultado final é algo do género:

```
grammar Calculator ;
program : stat* EOF ;
stat : expr NEWLINE    { System.out.println($expr.v); }
     | NEWLINE
     ;
expr returns[double v]
     : left=expr op=( '*' | '/' ) right=expr  { if($op.text.equals("*"))
                                                  $v = $left.v * $right.v;
                                                else $v = $left.v / $right.v; }
     | left=expr op=( '+' | '-' ) right=expr  { if($op.text.equals("+"))
                                                  $v = $left.v + $right.v;
                                                else $v = $left.v - $right.v; }
     | INT             { $v = Double.parseDouble($INT.text); }
     | '(' e=expr ')'  { $v = $e.v; }  // ou: '(' expr ')' { $v = $expr.v; }
     ;
INT : [0-9]+ ;
NEWLINE : '\r'? '\n' ;
WS: [ \t]+ -> skip ;
```

Note que multiplicações e divisões são feitas antes de somas e subtrações porque essa alternativa da regra `expr` aparece primeiro. Depois de compilar esta nova versão da gramática, a execução do comando

```
$ echo -e "2+3*4+5\n1+2*(3+2)/5-14" | java CalculatorMain
```

deve produzir `19.0` numa linha e `-11.0` na linha seguinte. Note que os resultados só são impressos quando se chega ao fim (`EOF`). Para evitar isso podemos criar a classe `CalculatorMain` usando o comando

```
$ antlr4-main -i Calculator program
```

A opção `-i` indica que se pretende separar o texto de entrada por linhas, sendo para cada uma delas desencadeado todo o processo de análise sintática (é como se cada linha fosse um ficheiro completo, com `EOF` e tudo). Isso pode levantar alguns problemas se for preciso usar informação numa linha que foi calculada numa linha anterior (ver solução do exercício 2.04).

Exercício 2.04

Implemente uma gramática para fazer a análise sintática dos ficheiros utilizados no exercício 1.3. Utilizando um *listener*, altere a resolução desse exercício por forma a incluir esta gramática na solução.

Relembra-se que no exercício 1.3 se pretendia traduzir, palavra a palavra, todas as ocorrências por extenso de números pelo respetivo valor numérico, mantendo todas as restantes palavras. A correspondência entre números e valores numéricos era feita de acordo com um tabela (dicionário) armazenada no ficheiro `numbers.txt`. Apresentamos a seguir as duas primeiras linhas desse ficheiro.

```
0 - zero
1 - one
```

No exercício 2.03 vamos usar um *listener* de uma gramática que vamos construir para verificar se esse ficheiro está formatado corretamente e para armazenar toda a informação nele contida. Note que cada linha do ficheiro `numbers.txt` é da forma `número - palavra`. O resto da resolução do exercício é semelhante à do exercício 1.3.

Proposta de solução. O primeiro passa para resolver este exercício consiste em construir a gramática, o que neste caso é muito simples:

```
grammar Numbers ;
file : line* EOF ;         // o nosso ficheiro é composto por zero ou mais linhas
line : NUM '-' WORD NL ;   // cada linha tem a forma NUM - WORD e termina com uma mudança
                           //   de linha
NUM  : [0-9]+ ;            // cada NUM é composto por 1 ou mais algarismos
WORD : [A-Za-z]+ ;         // cada WORD é composta por 1 ou mais letras
NL   : '\r'? '\n' ;
WS   : [ \t]+ -> skip ;
```

Esta gramática deve ser guardada no ficheiro `Numbers.g4` (o nome do ficheiro tem de ser o nome da gramática, com a extensão `.g4`). Para construir o esqueleto do nosso programa basta executar os comandos

```
$ antlr4 Numbers.g4
$ antlr4-main Numbers file
```

Para resolver este exercício são relevantes os ficheiros `NumbersListener.java` e `NumbersMain.java` (a primeira parte dos nomes destes ficheiros é o nome da gramática). No primeiro ficheiro é definido o interface do nosso *listener*. Em vez de escrever código diretamente neste ficheiro, é mais seguro criar uma outra classe que extende a classe que o `ANTLR` criou (deste modo, se a gramática for alterada e se o comando `antlr4` for corrido novamente, não perdemos nenhum código), fazendo o *override* dos métodos que precisamos de usar para resolver o nosso problema.

No nosso caso, a classe `NumbersListener.java` tem métodos que são invocados sempre que se entra a sempre que se sai de cada uma das regras da gramática. Esses métodos têm nomes relacionados com o nome das regras: `enterFile` e `exitFile`, e `enterLine` e `exitLine`. Estamos interessados no método `exitLine`, que é invocado para cada linha válida do nosso ficheiro `numbers.txt`. Na classe que vamos construir temos de guardar, tal como no exercício 1.3, todas as correspondências entre números e o seu texto por extenso numa estrutura de dados apropriada (um `Map` implementado com um `HashMap`). Chamando à nossa classe `ConstructNumbersMappings`, uma maneira possível de fazer tudo que é preciso é a seguinte:

```
import java.util.Map;
import java.util.HashMap;

public class ConstructNumberMappings extends NumbersBaseListener
{
  protected Map<String,Integer> mappings = new HashMap<String,Integer>();

  public boolean exists(String key)
  {
    assert key != null;
    return mappings.containsKey(key);
  }

  public Integer value(String key)
  {
    assert key != null;
    assert exists(key);
    return mappings.get(key);
  }

  @Override public void exitLine(NumbersParser.LineContext ctx)
  {
    String key = ctx.WORD().getText();
    Integer value = Integer.parseInt(ctx.NUM().getText());
    if(exists(key))
    {
      System.err.println("ERROR: repeated key \""+key+"\"");
      System.exit(1);
    }
    mappings.put(key,value);
  }
}
```

Note que a nossa classe tem um membro do tipo `Map` que é onde vamos guardar a informação recolhida do ficheiro `numbers.txt`. Além do método `exitLine` (a parte do *listener* que nos interessa), tem também dois métodos, a que chamámos `exists` e `value`, que nos permite extrair informação do nosso `Map`.

Quando à nossa classe principal, `NumbersMain`, vai ser preciso i) criar um `InputStream` para `numbers.txt` que vai ser usado em vez do `System.in` pela maquinaria do `ANTLR`; em vez de

```
   // create a CharStream that reads from standard input:
   ANTLRInputStream input = new ANTLRInputStream(System.in);
```

vamos ter, por exemplo,

```
   InputStream in_stream = null;
   try
   { in_stream = new FileInputStream(new File("numbers.txt")); }
   catch(FileNotFoundException e)
   { err.println("ERROR: reading number file!"); System.exit(1); }
   // create a CharStream that reads from in_stream:
   ANTLRInputStream input = new ANTLRInputStream(in_stream);
```

ii) se não existirem erros sintáticos, percorrer toda a árvore sintática (para que o nosso *listener* seja ativado nos instantes apropriados), por exemplo,

```
   // System.out.println(tree.toStringTree(parser));
   ParseTreeWalker walker = new ParseTreeWalker();
   ConstructNumberMappings mappings = new ConstructNumberMappings();
   walker.walk(mappings,tree);
```

e iii) finalmente processar o *standard input*, tal como no exercício 1.3. Disto tudo resulta algo do género:

```
import static java.lang.System.*;
import java.util.Scanner;
import java.io.*;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class NumbersMain {
  public static void main(String[] args) throws Exception {
    InputStream in_stream = null;
    try
      { in_stream = new FileInputStream(new File("numbers.txt")); }
    catch(FileNotFoundException e)
      { err.println("ERROR: reading number file!"); exit(1); }

    // create a CharStream that reads from in_stream:
    ANTLRInputStream input = new ANTLRInputStream(in_stream);
    // create a lexer that feeds off of input CharStream:
    NumbersLexer lexer = new NumbersLexer(input);
    // create a buffer of tokens pulled from the lexer:
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    // create a parser that feeds off the tokens buffer:
    NumbersParser parser = new NumbersParser(tokens);
    // begin parsing at file rule:
    ParseTree tree = parser.file();
    if(parser.getNumberOfSyntaxErrors() == 0) {
       // print LISP-style tree:
       // System.out.println(tree.toStringTree(parser));
       // listeners:
       ParseTreeWalker walker = new ParseTreeWalker();
       ConstructNumberMappings mappings = new ConstructNumberMappings();
```

```
        walker.walk(mappings,tree);

        Scanner in = new Scanner(System.in);
        while(in.hasNextLine())
        {
           String line = in.nextLine();
           String[] words = line.replace('-',' ').toLowerCase().split(" ");
           for(int i = 0;i < words.length;i++)
           {
              if(i != 0) out.print(" ");
              if(mappings.exists(words[i]))
                 out.print(mappings.value(words[i]).toString());
              else
                 out.print(words[i]);
           }
           out.println();
        }
     }
  }
}
```

Exercício 2.05

Altere o exercício 2.02, acrescentando a possibilidade de definir e utilizar variáveis. Para esse fim considere uma nova instrução

```
assignment : ID '=' expr NEWLINE ;
...
ID : [A-Za-z_]+ ;
```

Para dar suporte ao registo dos valores associados a variáveis, utilize, tal como no exercício 2.03, um *array* associativo (`HashMap`). Note que na definição da gramática pode-se indicar código a ser colocado no preâmbulo do código gerado (util para fazer *import*s), e podem ser indicados novos membros a adicionar às classes geradas, usando as construcões `@header{ ... }` e `@members { ... }`.

Proposta de solução. As alterações a efetuar em relação à solução do exercício 2.02 são simples:

```
grammar Calculator ;
@header { import java.util.*; }
@members { protected Map<String,Double> symbolTable = new HashMap<String,Double>(); }

program : ( stat {System.out.println($stat.v);} NEWLINE) * EOF
        ;
stat returns[double v]
     : expr       { $v = $expr.v; }
     | assignment { $v = $assignment.v; }
     ;
assignment returns[double v]
           : ID '=' expr  { $v = $expr.v; symbolTable.put($ID.text,$v); }
           ;
expr returns[double v]
     : left=expr op=( '*' | '/' ) right=expr  { if($op.text.equals("*"))
                                                   $v = $left.v * $right.v;
                                                else $v = $left.v / $right.v; }
     | left=expr op=( '+' | '-' ) right=expr  { if($op.text.equals("+"))
```

```
                                                    $v = $left.v + $right.v;
                                                else $v = $left.v - $right.v; }
    | INT           { $v = Double.parseDouble($INT.text); }
    | '(' expr ')'  { $v = $expr.v; }
    | ID              { if(!symbolTable.containsKey($ID.text))
                        { System.err.println("ERROR: variable \""+$ID.text+
                                             "\" not found!"); System.exit(1); }
                      $v = symbolTable.get($ID.text); }
    ;
ID : [A-Za-z_]+ ;
INT : [0-9]+ ;
NEWLINE : '\r'? '\n' ;
WS: [ \t]+ -> skip ;
```

Exercício 2.06

Descarregue a gramática da versão 8 da linguagem `Java` do repositório de gramáticas do `ANTLR` (https://raw.githubusercontent.com/antlr/grammars-v4/master/java8/Java8.g4) e experimente fazer a análise sintática de programas `Java` simples. Utilize o suporte para *listeners* do `ANTLR` para escrever o nome das classes e dos métodos sujeitos a análise sintática. Como extra, escreva também o nome dos campos (também conhecidos por atributos ou por, em Inglês, *fields*).

Proposta de solução. Como vamos usar um *listener*, temos em primeiro lugar de decidir que nome é que vamos dar à classe, feita por nós, que vai extrair a informação que pretendemos. Vamos usar o nome `ExtractInformation`. Sendo assim, começamos por gerar o esqueleto do código `Java` que vamos usar:

```
$ antlr4 -listener Java8.g4
$ antlr4-main Java8 compilationUnit -l ExtractInformation
$ javac *.java
```

(Como ainda não escrevemos o código para a classe `ExtractInformation` a compilação da classe `Java8Main` dá erro, mas tudo o resto é compilado corretamente.) Note que o nome da regra de entrada, `compilationUnit`, aparece como comentário no início do ficheiro `Java8.g4`. Note ainda que numa gramática bem feita a regra de entrada é a **única** que acaba com `EOF`. Depois de se ter feito isto tudo a classe `Java8Main` já fica com a infraestrutura apropriada para invocar o nosso *listener*. Falta agora criar o código fonte da classe `ExtractInformation`. Aqui, o problema é descobrir os nomes dos métodos da classe `Java8BaseListener`, criada pelo `ANTLR`, que vamos substituir usando o mecanismo de *override* do `Java`. Para o efeito criamos um ficheiro de teste, a que vamos dar o nome `Test.java` e onde colocamos, por exemplo, o código

```
public class Test
{
   public int f(int z) { return z; }
   public int x;
   public double p(double x) { return x + 1.0; }
}
```

Depois é só estudar a árvore sintática que é gerada quando este ficheiro é analizado pelo *parser* gerado pelo `ANTLR`, obtida usando o comando

```
$ cat Test.java | antlr4-test Java8 compilationUnit -gui
```

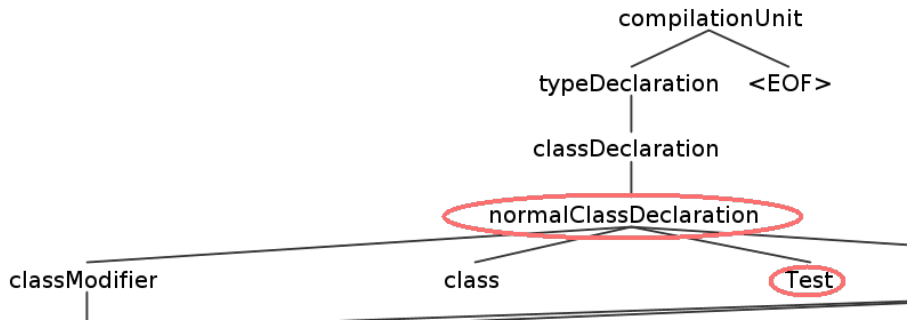

Figure 1: Parte da árvore sintática do ficheiro `Test.java`.

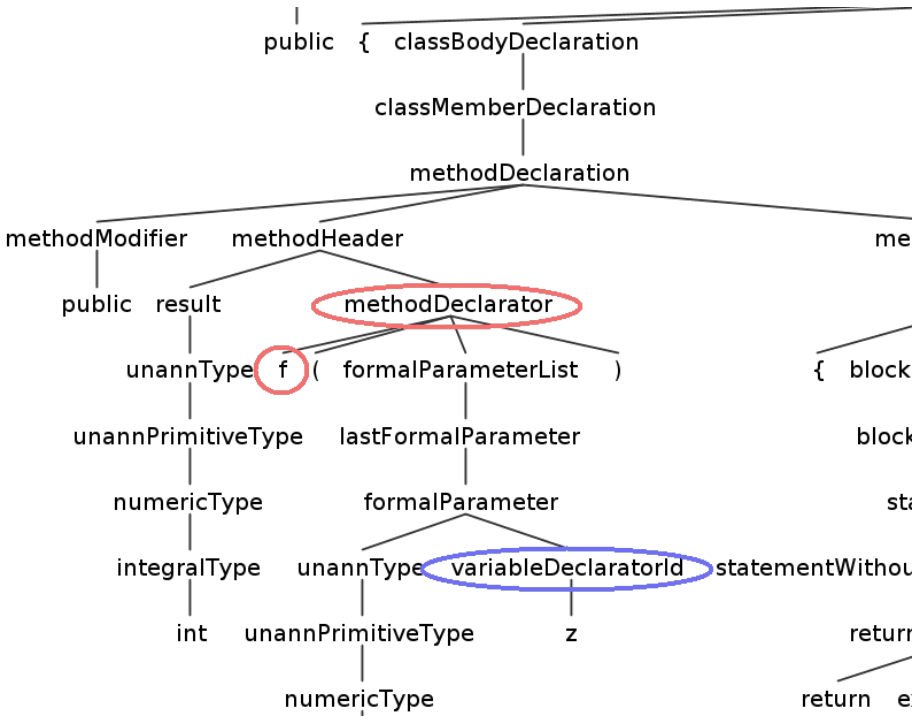

Figure 2: Outra parte da árvore sintática do ficheiro `Test.java`.

Analizando a árvore sintática que é mostrada verifica-se que o nome da nossa classe (`Test`) se encontra pendurado num nó da árvore com o nome `normalClassDeclaration` (ver figura 1). Analizando essa regra da gramática (isto é, dado uma olhadela ao conteúdo do ficheiro `Java8.g4`), constatamos que o nome da classe corresponde à regra `Identifier`. No ficheiro `Java8BaseListener.java` encontramos a declaração completa do método que queremos alterar (`enter` mais `normalClassDeclaration` dá um método com o nome `enterNormalClassDeclaration`). Munidos desta informação toda estamos agora em condições de escrever o nosso primeiro método:

```
@Override
public void enterNormalClassDeclaration(Java8Parser.NormalClassDeclarationContext ctx)
{
  out.println("class "+ctx.Identifier().getText());
}
```

Quanto aos nomes dos métodos da classe `Test`, eles encontram-se pendurados em nós com o nome `methodDeclarator` (ver figura 2), pelo que o código para os imprimir é muito parecido ao anterior.

Finalmente, para imprimir os nomes dos campos, temos de olhar para outra parte da árvore sintática (ver figura 3). Aqui a situação é mais complicada porque o nome do campo (no nosso caso `x`) encontra-se pendurado num nó com o nome `variableDeclaratorId`, e outros nós da árvore também têm esse nome (por exemplo, aparecem nos argumentos dos métodos). Uma maneira de resolver este

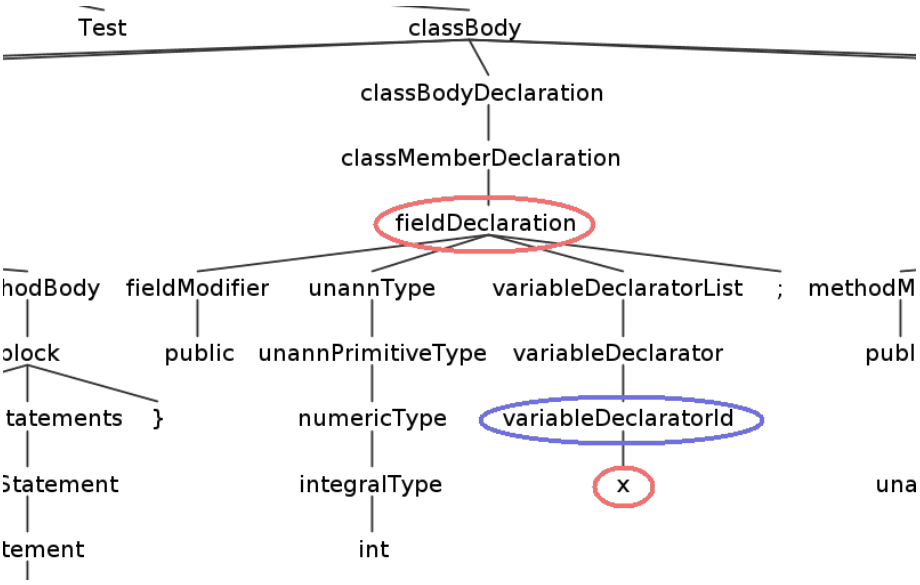

Figure 3: Ainda outra parte da árvore sintática do ficheiro `Test.java`.

problema consiste em registar as entradas e saídas nos nós com o nome `fieldDeclaration`, e só imprimir um `variableDeclaratorId` quando este aparece dentro de um `fieldDeclaration`.

O código seguinte é uma solução completa do exercício (ficheiro `ExtractInformation.java`):

```
public class ExtractInformation extends Java8BaseListener
{
  @Override
  public void enterNormalClassDeclaration(Java8Parser.NormalClassDeclarationContext ctx)
  {
    System.out.println("class "+ctx.Identifier().getText());
  }
  @Override
  public void enterMethodDeclarator(Java8Parser.MethodDeclaratorContext ctx)
  {
    System.out.println(" method "+ctx.Identifier().getText());
  }
  protected boolean inside = false;
  @Override
  public void enterFieldDeclaration(Java8Parser.FieldDeclarationContext ctx)
  {
    inside = true;
  }
  @Override
  public void exitFieldDeclaration(Java8Parser.FieldDeclarationContext ctx)
  {
    inside = false;
  }
  @Override
  public void enterVariableDeclaratorId(Java8Parser.VariableDeclaratorIdContext ctx)
  {
    if(inside)
      System.out.println(" field  "+ctx.Identifier().getText());
  }
}
```

Exercício 2.07

Utilizando a gramática definida no exercício 2.04 e recorrendo aos *visitors* do `ANTLR`, converta uma expressão aritmética infixa (operador no meio dos operandos) numa expressão equivalente sufixa (operador no fim, também conhecida em Inglês por *Reverse Polish Notation*). Por exemplo,

$$\begin{array}{ll} 2+3 & \rightarrow \ 2\ 3\ + \\ 2+3*4 & \rightarrow \ 2\ 3\ 4\ *\ + \\ 3*(2+1)+(2-1) & \rightarrow \ 3\ 2\ 1\ +\ *\ 2\ 1\ -\ + \end{array}$$

Em `ANTLR` podemos associar diferentes *callbacks* a diferentes alternativas numa regra sintática:

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

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

Proposta de solução. Usando a gramática do exercício 2.04 como ponto de partida, temos de dar nomes diferentes às várias alternativas das regras que as têm. Como o objetivo não é neste caso fazer as contas podemos ou eliminar todo o código `Java` especificado na gramática ou comentar os `println`. A primeira possibilidade é mais limpa, pelo que é isso que fazemos aqui:

```
grammar Calculator;
program : ( stat NEWLINE ) * EOF
        ;

stat : expr       # ExprStat
     | assignment # AssignmentStat
     ;

assignment : ID '=' expr
           ;

expr : e1=expr op=('*'|'/') e2=expr # MultDiv
     | e1=expr op=('+'|'-') e2=expr # AddSub
     | INT                         # Number
     | '(' expr ')'                # Parentesis
     | ID                          # Variable
     ;

ID: [a-zA-Z_]+;
INT: [0-9]+;
NEWLINE: '\r'? '\n';
WS: [ \t]+ -> skip;
```

Dando o nome de `ConvertToSuffix` à nossa classe que vai efetuar a conversão, o `ANTLR` e o *script* `antlr4-main` fazem uma boa parte do trabalho:

```
$ antlr4 -visitor -no-listener Calculator.g4
$ antlr4-main Calculator.g4 program -v ConvertToSuffix.java -i
```

A opção `-i` é necessária para se ver o resultado da conversão linha a linha. No nosso caso o resultado

da nossa visita da árvore sintática é uma *string*. Para a ver a linha de código

```
visitor0.visit(tree);
```

do ficheiro `CalculatorMain.java` tem de ser substituida por

```
System.out.println(visitor0.visit(tree));
```

ou, melhor ainda, por

```
System.out.printf("%s  -->  %s",lineText,visitor0.visit(tree));
```

Como os *visitors* devolvem *strings*, a nossa classe (ficheiro `ConvertToSuffix.java`) terá a seguinte forma

```
public class ConvertToSuffix extends CalculatorBaseVisitor<String>
{ // ...
}
```

O primeiro método *visitor* que vamos fazer é o para a regra principal. Neste caso esta regra é a mais complicada para se fazer um *visitor* porque o nó correpondente da árvore sintática pode ter um número variável de descendentes. O *ANTLR* guarda essa informação numa lista, pelo que para os visitar a todos e processar o valor que cada um retorna teremos de usar um iterador. O código seguinte mostra uma maneira de fazer isso:

```
@Override
public String visitProgram(CalculatorParser.ProgramContext ctx)
{
  String res = "";
  Iterator<CalculatorParser.StatContext> iter = ctx.stat().iterator();
  while(iter.hasNext())
    res += visit(iter.next()) + "\n";
  return res;
}
```

Note que o nome do método é a junção de `visit` com o nome da regra (com a primeira letra transformada em maiúscula), e que o nome do tipo do seu argumento é construido juntando o nome da gramática a `Parser`, seguido de um ponto, seguido pelo nome da regra (mais uma vez com a primeira letra transformada em maiúscula), a que se junta `Context`. Apreveitamos para mencionar aqui que algo semelhante se passa igualmente para os *listeners*. Finalmente, é importante mencionar que o nome do nó a visitar, que neste caso corresponde à regra `stat`, pode em geral ser ou um método ou um campo. Neste caso é um método, pelo no nosso código escrevemos `visit(ctx.stat())`. Se fosse um campo seria apenas `visit(ctx.stat)`.

Nas duas alternativas da regra `stat` é apenas preciso visitar os seus descendentes:

```
@Override
public String visitExprStat(CalculatorParser.ExprStatContext ctx)
{
  return visit(ctx.expr());
}
@Override
public String visitAssignmentStat(CalculatorParser.AssignmentStatContext ctx)
{
  return visit(ctx.assignment());
}
```

Repare que os nomes dos nós a visitar são os nomes das respetivas regras.

A regra `assignment` é a primeira onde de facto se passa alguma coisa interessante. No código que se segue optámos por returnar o resultado na forma `ID expr =`, sendo obviamente `expr` convertido numa expressão sufixa:

```
  @Override
  public String visitAssignment(CalculatorParser.AssignmentContext ctx)
  {
    return ctx.ID().getText() + " " + visit(ctx.expr()) + " =";
  }
```

Exatamente a mesma estratégia pode ser usada para as alternativas `MultDiv` e `AddSub` da regra `expr`:

```
  @Override
  public String visitMultDiv(CalculatorParser.MultDivContext ctx)
  {
    return visit(ctx.e1) + " " + visit(ctx.e2) + " " + ctx.op.getText();
  }
  @Override
  public String visitAddSub(CalculatorParser.AddSubContext ctx)
  {
    return visit(ctx.e1) + " " + visit(ctx.e2) + " " + ctx.op.getText();
  }
```

Note que aqui, como foram dados os nomes `e1`, `op` e `e2` às várias partes destas variante da regra, escrevemos `visit(ctx.e1)` em vez de `visit(ctx.e1())` — é um campo — o mesmo se passando em relação a `op` e a `e2`. As restantes variantes desta regra não oferecem nenhuma dificuldade, pelo que as apresentamos conjuntamente por o resto do código (conteúdo completo do ficheiro `ConvertToSuffix.java`:

```
import java.util.*;

public class ConvertToSuffix extends CalculatorBaseVisitor<String>
{
  @Override
  public String visitProgram(CalculatorParser.ProgramContext ctx)
  {
    String res = "";
    Iterator<CalculatorParser.StatContext> iter = ctx.stat().iterator();
    while(iter.hasNext())
      res += visit(iter.next()) + "\n";
    return res;
  }
  @Override
  public String visitExprStat(CalculatorParser.ExprStatContext ctx)
  {
    return visit(ctx.expr());
  }
  @Override
  public String visitAssignmentStat(CalculatorParser.AssignmentStatContext ctx)
  {
    return visit(ctx.assignment());
  }
  @Override
  public String visitAssignment(CalculatorParser.AssignmentContext ctx)
```

```
  {
    return ctx.ID().getText() + " " + visit(ctx.expr()) + " =";
  }
  @Override
  public String visitMultDiv(CalculatorParser.MultDivContext ctx)
  {
    return visit(ctx.e1) + " " + visit(ctx.e2) + " " + ctx.op.getText();
  }
  @Override
  public String visitAddSub(CalculatorParser.AddSubContext ctx)
  {
    return visit(ctx.e1) + " " + visit(ctx.e2) + " " + ctx.op.getText();
  }
  @Override
  public String visitNumber(CalculatorParser.NumberContext ctx)
  {
    return ctx.INT().getText();
  }
  @Override
  public String visitParentesis(CalculatorParser.ParentesisContext ctx)
  {
    return visit(ctx.expr());
  }
  @Override
  public String visitVariable(CalculatorParser.VariableContext ctx)
  {
    return ctx.ID().getText();
  }
}
```

Exercício 2.09

Pretende-se desenvolver uma calculadora simples para operações sobre conjuntos. Vamos limitar os conjuntos a listas finitas de elementos, sendo cada elemento do conjunto ou um número ou uma palavra. Desenvolva uma gramática para esta calculadora, tendo em consideração a seguinte especificação:

- Um conjunto é definido por uma sequência de palavras, ou de números, separada por vírgulas e delimitada por chavetas. Por exemplo, $\{a, b, c\}$, ou $\{1, -3, 5, 7, 9\}$.
- 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. Por exemplo, $C = \{a, b, c\}$.
- Uma variável é uma sequência de letras maiúsculas.
- Implemente as operações (com prioridade crescente) sobre conjuntos: união (definida pelo símbolo +), interseção (símbolo &) e diferença (símbolo \). Por exemplo, $\{a, b, c\} + \{b, d\}$.
- Para poder definir diferentes precedências, implemente os parêntesis. Por exemplo, $(\{a\} + \{b\})\backslash\{b\}$.
- Implemente comentários de uma linha definidos pelo prefixo `--`.[1] Por exemplo,

[1] VHDL *rules*! O autor deste documento não tem nada a ver com esta especificação.

```
-- 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. Por exemplo

$$
\begin{array}{lcl}
C = \{a, b, c\} & \rightarrow & \{a, b, c\} \\
\texttt{-- comment} & \rightarrow & \\
\{a, b, c\} + \{b, d\} & \rightarrow & \{a, b, c, d\} \\
\{a, b, c\} \,\&\, \{b, d\} & \rightarrow & \{b\} \\
C \setminus \{b, d\} & \rightarrow & \{a, c\} \\
C \setminus C & \rightarrow & \{\}
\end{array}
$$

Proposta de solução. A primeira coisa a fazer é responder à pergunta: vamos acrescentar código `Java` à nossa gramática, vamos usar *listeners*, vamos usar *visitors*, ou vamos misturar mais do que uma desses técnicas? Apesar de essa não ser a única solução possível, vamos aqui, tal como no exercício 2.06, usar apenas *visitors*. Sendo assim, uma especificação possível da gramática, a que vamos chamar `SetCalc`, é

```
grammar SetCalc;

main : stat * EOF
     ;

stat : expr
     ;

expr : e1=expr '\\' e2=expr # ExprSubtract
     | e1=expr '&' e2=expr  # ExprIntersect
     | e1=expr '+' e2=expr  # ExprUnion
     | '(' e=expr ')'       # ExprParen
     | set                  # ExprSet
     | VAR '=' e=expr       # ExprAssign
     | VAR                  # ExprVar
     ;

set : '{' ( elem ( ',' elem )* )? '}'
    ;

elem : WORD | NUM
     ;

WORD    : [a-z]+                ;
NUM     : [-+]?[0-9]+           ; // . means any character
VAR     : [A-Z]+                ; // .* means any number of characters
COMMENT : '--' .*? '\n' -> skip ; // .*? means any number of characters that to not
WS      : [ \t\r\n]+ -> skip    ; //   match what comes next
ERROR   : .                     ; // to convert all lexer errors into parser errors
```

Chamando à nossa classe `Compute`, criamos os nosso ficheiros como de costume:

```
$ antlr4 -visitor -no-listener SetCalc.g4
$ antlr4-main SetCalc.g4 main -v Compute.java
```

Levanta-se agora o problema de que estrutura de dados usar para representar um conjunto. Atendendo a que os elementos do conjunto podem ser palavras ou números, para simplificar o problema

vamos usar para esse efeito um conjunto de *strings*: `Set<String>`. Em particular, os nossos números também vão ser *strings*. É pois este tipo de dados que os métodos da nossa classe vão devolver. A classe tem, no entanto, de ter um campo onde se vai guardar o último valor de todas as variáveis que vão sendo inicializadas. Para isso vamos usar um `Map<String,Set<String>>`, que nos permite associar um conjunto a uma *string* (com o nome de uma variável). O código inicial a colocar na nossa classe será então:

```
import static java.lang.System.*;
import java.util.*;

public class Compute extends SetCalcBaseVisitor<Set<String>>
{
  protected Map<String,Set<String>> varTable = new HashMap<String,Set<String>>();
  // ...
}
```

Para evitar o iterador usado na regra de entrada do exercício anterior, na gramática que fizemos para este exercício foi criada de propósito uma regra, com o nome `stat`, que é o sítio onde vamos imprimir o resultado de cada expressão. Como estamos a lidar com conjuntos de *strings*, o código do seu *visitor* pode ser algo do género:

```
  @Override
  public Set<String> visitStat(SetCalcParser.StatContext ctx)
  {
    Set<String> result = visit(ctx.expr());
    String str = result.toString();
    str = "{" + str.substring(1,str.length()-1).replaceAll(" ","") + "}";
    out.println("result: " + str);
    return result;
  }
```

Para o resto dos métodos a maneira de proceder é semelhante à do exercício anterior. A maior dificuldade tem a ver com a implementação das operações com conjuntos (ver código abaixo). O último problema que temos de resolver diz respeito à maneira como vamos construir um conjunto (regra `set`) a partir dos seus elementos (regra `elem`). Uma maneira simples consiste em definir um campo que guarda um conjunto temporário na nossa classe, a que vamos dar o nome `tempSet`, que é inicializado no início do código do *visitor* da regra `set`, que é atualizado no *visitor* da regra `elem` e que é depois utilizado como valor a devolver pelo *visitor* da regra `set`. O código correspondente é algo do género:

```
  protected Set<String> tempSet;
  @Override
  public Set<String> visitSet(SetCalcParser.SetContext ctx)
  {
    tempSet = new HashSet<String>();
    visitChildren(ctx);
    return tempSet;
  }
  @Override
  public Set<String> visitElem(SetCalcParser.ElemContext ctx)
  {
    tempSet.add(ctx.getText());
    return null;
  }
```

A maneira simplificada como as coisas estão a ser feitas aqui (não chegamos a calcular o valor numérico dos números), tem como consequência indesejada que `007` é diferente de apenas `7`. Apresentamos de seguida o código completo da class `Compute`:

```
import static java.lang.System.*;
import java.util.*;

public class Compute extends SetCalcBaseVisitor<Set<String>>
{
  protected Map<String,Set<String>> varTable = new HashMap<String,Set<String>>();
  @Override
  public Set<String> visitStat(SetCalcParser.StatContext ctx)
  {
    Set<String> result = visit(ctx.expr());
    String str = result.toString();
    str = "{" + str.substring(1,str.length()-1).replaceAll(" ","") + "}";
    out.println("result: " + str);
    return result;
  }
  @Override
  public Set<String> visitExprSubtract(SetCalcParser.ExprSubtractContext ctx)
  {
    Set<String> res = new HashSet<String>();
    Set<String> s1 = visit(ctx.e1);
    Set<String> s2 = visit(ctx.e2);
    Iterator<String> iter = s1.iterator();
    while(iter.hasNext())
    {
      String e = iter.next();
      if(!s2.contains(e))
        res.add(e);
    }
    return res;
  }
  @Override
  public Set<String> visitExprIntersect(SetCalcParser.ExprIntersectContext ctx)
  {
    Set<String> res = new HashSet<String>();
    Set<String> s1 = visit(ctx.e1);
    Set<String> s2 = visit(ctx.e2);
    Iterator<String> iter = s1.iterator();
    while(iter.hasNext())
    {
      String e = iter.next();
      if(s2.contains(e))
        res.add(e);
    }
    return res;
  }
  @Override
  public Set<String> visitExprUnion(SetCalcParser.ExprUnionContext ctx)
  {
    Set<String> res;
    Set<String> s1 = visit(ctx.e1);
    Set<String> s2 = visit(ctx.e2);
```

```
    res = new HashSet<String>(s1);
    Iterator<String> iter = s2.iterator();
    while(iter.hasNext())
      res.add(iter.next());
    return res;
  }
  @Override
  public Set<String> visitExprParen(SetCalcParser.ExprParenContext ctx)
  {
    return visit(ctx.e);
  }
  @Override
  public Set<String> visitExprSet(SetCalcParser.ExprSetContext ctx)
  {
    return visit(ctx.set());
  }
  @Override
  public Set<String> visitExprAssign(SetCalcParser.ExprAssignContext ctx)
  {
    Set<String> res = visit(ctx.e);
    String var = ctx.VAR().getText();
    varTable.put(var, res);
    return res;
  }
  @Override
  public Set<String> visitExprVar(SetCalcParser.ExprVarContext ctx)
  {
    Set<String> res = null;
    String var = ctx.VAR().getText();
    if(!varTable.containsKey(var))
    {
      err.println("ERROR: variable \"" + var + "\" not defined!");
      exit(1);
    }
    res= varTable.get(var);
    return res;
  }
  protected Set<String> tempSet;
  @Override
  public Set<String> visitSet(SetCalcParser.SetContext ctx)
  {
    tempSet = new HashSet<String>();
    visitChildren(ctx);
    return tempSet;
  }
  @Override
  public Set<String> visitElem(SetCalcParser.ElemContext ctx)
  {
    tempSet.add(ctx.getText());
    return null;
  }
}
```