Compiladores / Linguagens Formais e Autómatos
Resolução detalhada do exercício do 2.1

Artur Pereira, Tomás Oliveira e Silva, e Miguel Oliveira e Silva

Março de 2020

Exercício 2.1

Crie o ficheiro `Hello.g4` com 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.

Solução esquartejada.

Num terminal, no diretório onde se encontra o ficheiro `Hello.g4`, execute o comando

```
antlr4 Hello.g4
```

Se não ocorrerem erros, este comando vai criar vários ficheiros `.java`. Em particular, são criados o *lexer* e o *parser* para a gramática especificada no ficheiro `Hello.g4`.

A seguir execute o comando

```
antlr4-build
```

Este comando vai compilar todos os ficheiros `.java` existentes no diretório corrente. (Na realidade, este comando invoca também o `antlr4`, pelo que bastaria executar este.)

A seguir vai usar-se o *TestRig* do `ANTLR` para a gramática `Hello` usando `greetings` como símbolo inicial, passando-lhe uma expressão válida sintaticamente. Para o fazer, execute, por exemplo, o comando

```
echo "hello aluna" | antlr4-test Hello greetings
```

Se tudo estiver em ordem, não é produzido nenhum texto. Pode-se 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.

Veja-se agora o resultado no caso de uma expressão errada sintaticamente. Para o fazer, execute, por exemplo, o comando

```
echo "Hello aluna" | antlr4-test Hello greetings
```

Neste caso, deve aparecer no ecrã uma mensagem de erro, dizendo que aparece `Hello` onde deveria aparecer `hello`.

Em todo o processo anterior não foi gerado um programa baseado na gramática `Hello`. Para isso pode usar-se o comando

```
antlr4-main Hello greetings
```

Este comando gera código `Java` que define uma classe *main* 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 do símbolo inicial (normalmente o primeiro) da gramática.

Depois de compilado (`antrl4-build`), pode ser executado, fornecendo-lhe como entrada a expressão (texto) que será avaliada sintaticamente, através do comando

```
echo "hello aluna" | antlr4-run
```

ou do comando

```
echo "hello aluna" | java -ea HelloMain
```

Se tudo correr bem, não deve ser produzida qualquer saída.

(b) Acrescente um *visitor* (`Execute`) de modo a, no fim da regra `greeting: 'hello' ID`, escrever em português `Olá <ID>`, onde `<ID>` representa o nome que for colocado à frente de `'hello'` aquando da invocação do programa correspondente.

Solução esquartejada.

Primeiro, é necessário 'pedir' ao `antlr4` para gerar o suporte para *visitors*. Para isso, execute o comando

```
antlr4 Hello.g4 -visitor
```

Além do referido anteriormente, são gerados os ficheiros `HelloVisitor.java`, um interface, e `HelloBaseVisitor.java`, uma implementação base desse interface. Este *visitor* base pode ser usado para criar o nosso, através do comando

```
antlr4-visitor Execute String
```

Este comando cria um *visitor*, chamado `Execute`, baseado na gramática `Hello`[1] e associado ao tipo de dados `String`. Note que o *visitor* base é um genérico.

A seguir é necessário criar um programa que use o *visitor* e editar esse *visitor* para pô-lo a fazer o pretendido. Começando pela edição, altere o ficheiro `Execute.java`, fazendo o corpo do método `visitGreetings` igual a

```
    System.out.println("Olá " + ctx.ID().getText());
    return null;
```

O contexto da produção `greetings: 'hello' ID` inclui o símbolo terminal `ID` que, em cada execução corresponde à palavra colocada a seguir ao `'hello'`. A *string* representando essa palavra pode ser obtida pela expressão `ctx.ID().getText()`.

Para gerar o programa que chama o *visitor* `Execute` pode-se executar o comando

[1] Porque neste diretório só existe um ficheiro .g4.

```
antlr4-main Hello greetings -visitor Execute
```

Finalmente pode executá-lo e ver o resultado, por exemplo com

```
echo "hello aluna" | antlr4-run
```

(c) Acrescente uma produção de despedida à gramática (`bye : 'goodbye' ID`), fazendo com que a gramática aceite uma qualquer das regras (`greetings` ou `bye`).

Solução esquartejada.

Apresenta-se a seguir (parcialmente) a alteração proposta à gramática

```
top     :   greetings |   bye;
greetings   :   'hello' ID;
bye         :   'goodbye' ID;
...
```

Esta nova gramática introduz algumas mudanças no que tínhamos anteriormente. O símbolo inicial passou a ser `top` e não `greetings`. Isto obriga a que a classe *main* tenha de ser alterada. A alteração pode ser concretizada editando o ficheiro `HelloMain.java`, substituindo

```
    ParseTree tree = parser.greetings();
```

por

```
    ParseTree tree = parser.top();
```

ou gerando um novo

```
rm -f HelloMain.java
antlr4-main Hello top
```

A gramática passou a ter 3 símbolos não terminais (`top`, `greetings` e `bye`), pelo que é necessário gerar novos *visitors*. O interface e o *visitor* base são gerados como indicado anteriormente

```
antlr4 Hello.g4 -visitor
```

A alteração do `Execute.java` pode ser feita apagando-o e gerando um novo

```
antlr4-visitor Execute String
```

ou editando-o, incorporando os métodos em falta (`visitTop` e `visitBye`).

O método `visitTop` não precisa de ser alterado. O método `visitBye` deverá ficar semelhante ao `visitGreetings` da alínea anterior, com, por exemplo, `Adeus` em vez de `Olá`.

Pode testar agora as duas situações:

```
echo "hello aluna" | antlr4-run
echo "goodbye aluna" | antlr4-run
```

(d) Generalize os identificadores por forma a incluir letras maiúsculas e a permitir nomes com mais do que um identificador.

Solução esquartejada.

Apresenta-se a seguir uma proposta para a nova gramática

```
top     :   greetings |   bye;
greetings   :   'hello' name;
bye         :   'goodbye' name;
name        :   ID+;

ID :   [A-Za-z]+;
WS : [ \t\n\r]+ -> skip ;
```

Optou-se por criar um novo símbolo não terminal (`name`) que representa a sequência de identificadores. Abaixo perceber-se-á a vantagem desta decisão.

Como o símbolo inicial (`top`) se manteve, não é necessário alterar a classe *main*.

No entanto, há uma nova produção (`name : ID+`), o que significa alterações nos *vistors*. Gerando novamente o interface e o *visitor* base,

```
antlr4 Hello.g4 -visitor
```

pode constatar-se que há um novo método (`visitName`). Para alterar o ficheiro `Execute.java` propõe-se que, em vez de gerar um novo, este método seja acrescentado. Para garantir que não se cometem erros, pode gerar-se um novo *visitor* base, por exemplo

```
antlr4-visitor Execute2 String
```

e copiar-se o método `visitName` para o ficheiro `Execute.java`. Este método pode ser usado para construir e devolver uma `String` que represente a sequência de identificadores (`ID+`),

```
    import java.util.Iterator;
    import org.antlr.v4.runtime.tree.TerminalNode;
    ...
    @Override public String visitName(HelloParser.NameContext ctx) {
        Iterator<TerminalNode> iter = ctx.ID().iterator();
        String res = "";
        while (iter.hasNext())
        {
            res += iter.next() + " ";
        }
        return res.trim();
    }
```

A seguir, basta alterar os métodos `visitGreetings` e `visitBye` para usarem o valor retornado por este método.

```
        @Override public String visitGreetings(HelloParser.GreetingsContext ctx) {
        System.out.println("Olá \"" + visit(ctx.name()) + "\"");
        return null;
    }
        @Override public String visitBye(HelloParser.ByeContext ctx) {
        System.out.println("Adeus \"" + visit(ctx.name()) + "\"");
        return null;
    }
```

Se se tivesse optado por pôr `ID+` nas produções `greetings` e `bye`, o código colocado no `visitName` teria que ser repetido nos dois métodos anteriores.

Agora, basta compilar e executar.

```
antlr4-build
echo "hello Fulano de Tal e Qual" | antlr4-run
echo "bye Fulano de Tal e Qual" | antlr4-run
```

(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`).

Solução esquartejada.

Esta questão pode ser tratada a dois níveis. Se se pretender construir um interpretador que responda a cada linha, basta gerar uma nova classe *main* em modo interativo.

```
rm -f HelloMain.java
antlr4-main Hello top -visitor Execute -i
```

A nova classe *main* decompõe a entrada em linhas e aplica a gramática a cada uma. Pode experimentar.

```
antlr4-build
antlr4-run
hello Fulano de Tal e Qual
bye Mundo cruel
^D
```

A execução de `antlr4-run` fica à espera da inserção de linhas através do *standard imput* (teclado, neste caso). O `^D` corresponde a 'inserir' o `EOF`. O programa vai respondendo a cada linha inserida.

Se se pretender construir um programa que responda apenas no fim da entrada, pode alterar-se a gramática, acrescentando a produção

```
main    :   top* EOF;
```

e criando uma classe *main* em modo não interativo que use `main` como símbolo inicial.

```
rm -f HelloMain.java
antlr4-main Hello main -visitor Execute
antlr4-build
antlr4-run
hello Fulano de Tal e Qual
bye Mundo cruel
^D
```

Note que para a classe *main* em modo interativo esta última gramática também funciona corretamente.

```
rm -f HelloMain.java
antlr4-main Hello main -visitor Execute -i
antlr4-build
antlr4-run
hello Fulano de Tal e Qual
bye Mundo cruel
^D
```