Tema 2-E1

ANTLR4

Exemplo de aplicação a uma linguagem simples

Compiladores, 2º semestre 2021-2022

Miguel Oliveira e Silva, Artur Pereira, DETI, Universidade de Aveiro

Conteúdo

1 Exemplo figuras

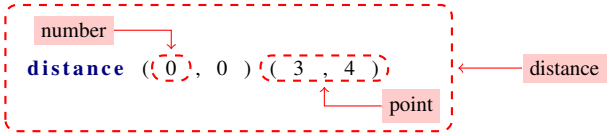
- O primeiro desafio é arranjar formas para definir linguagens.
- Em geral, as linguagens são construídas recorrendo a duas definições separadas: *léxica* e *sintáctica*.
- Na primeira define-se as regras para construir os símbolos léxicos (*tokens*) a partir da sequência de caracteres presente no código fonte da linguagem (palavras reservadas, identificadores, números literais, strings, etc.).
  Por exemplo: `ID: [a-z]+;`
- Na segunda, as regras que definem a estrutura sintáctica da linguagem (instruções, expressões, métodos, classes, etc.).
  Por exemplo: `assignment: ID '='expr;`
- Em ANTLR, as regras léxicas são construídas recorrendo essencialmente a *expressões regulares*, e as regras sintácticas a *gramáticas*.

- Na sua essência, o formalismo das gramáticas é uma forma de programação simbólica, em que estamos a definir símbolos (palavras escolhidas por nós), à custa de outros símbolos (ou mesmo do próprio, quando a recursividade faz sentido).

$$\alpha \rightarrow \beta$$

- Os símbolos presentes em β podem ser outros símbolos não terminais (i.e. a símbolos α definidos pela gramática), ou símbolos terminais (símbolos léxicos ou *tokens*).
- No fim, uma gramática para uma linguagem tem de permitir que, para qualquer programa dessa linguagem, começando no símbolo inicial (o que define a linguagem), se chegue a uma sequência de *tokens* idêntica à presente nesse programa, apenas por aplicação das respectivas regras sintácticas (i.e. substituindo sucessivamente os vários α por pelo menos um dos β prescritos na definição da linguagem).

- Considere o seguinte excerto de um programa:

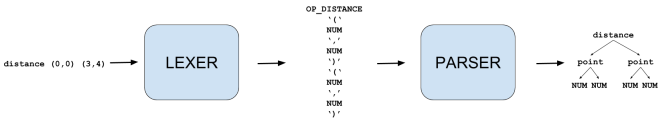


- Gramática inicial:

```
grammar Shapes;
// parser rules:
distance: 'distance' point point;
point: '(' x= Number ',' y= Number ')';
// lexer rules:
Number: [0-9]+;
WhiteSpace: [ \t\n\r]+ -> skip;
```

alias

- Para o programa exemplificado teríamos:



- Vamos programar esta gramática em ANTLR4 e testá-la com o script `antlr4-test`.
- Estando a gramática construída podíamos implementar um interpretador simples.
- No entanto, antes de fazermos esse exercício, vamos analisar mais criteriosamente a gramática apresentada, e fazer alguns melhoramentos.
- Desde logo, é fácil constatar que a linguagem definida por esta gramática é extremamente limitada, já que permite apenas uma (e só uma) instrução de distância entre dois pontos.
- Podemos rectificar este problema criando uma regra inicial que permita qualquer número de instruções:

```
...
main: stat* EOF;
stat: distance;
...
```

- Uma linguagem que apenas permite coordenadas com valores literais para definir pontos também representa uma limitação excessiva.
- Vamos introduzir uma nova regra – expressão – que representa um qualquer valor numérico escalar. Vamos para já considerar que uma expressão pode resultar dum número literal, ou duma operação aritmética elementar:

```
...
point: '(' x=expr ',' y=expr ')';
expr:
    expr op=('*'|'/') expr
  | expr op=('+'|'-') expr
  | '(' expr ')'
  | Number
  ;
...
```

- Agora já podemos ter programas um pouco mais interessantes:

```
distance ( 1+2*3-7 , 4-1-3 ) ( 3*2/2 , (1+1+2+3+4+5)/4 )
```

- Se uma expressão representa um valor escalar, porque não então aceitar também a distância como uma possível variante dessa regra:

```
...
expr:
    ...
  | distance
    ...
  ;
...
```

- Agora já podemos ter programas ainda mais interessantes:

```
distance (distance(1,1)(1,1), 4-1-3) (3*2/2 , (1+1+2+3+4+5)/4)
```

- Sendo a distância apenas uma variante entre várias alternativas de expressões, não faz muito sentido limitar as instruções da linguagem apenas a distâncias:

```
...
stat: expr;
...
```

- Estamos agora em melhores condições para passarmos à parte em que a linguagem ganha vida implementando (por exemplo) um interpretador para cada linha do código fonte.
- Temos basicamente três alternativas para essa implementação: utilizar um *visitor*, um *listener* ou injectando acções na gramática.
- Vamos começar pela primeira alternativa (que não só é a melhor como também a mais simples).

1.1 Exemplo *visitor*

- Uma primeira versão (limpa) de um *visitor* pode ser gerada com o script `antlr4-visitor`
- No entanto, antes de fazermos isso, vamos acrescentar à gramática etiquetas (*labels*) para as diferentes variantes das expressões:

```
...
expr:
    expr op=('*'|'/') expr #ExprMultDiv
  | expr op=('+'|'-') expr #ExprAddSub
  | '(' expr ')'           #ExprParenthesis
  | distance               #ExprDistance
  | Number                 #ExprNumber
  ;
...
```

- Desta forma, a classe gerada para um *visitor* incluirá diferentes métodos para cada uma das alternativas (evitando a necessidade de repetir uma parte da análise sintáctica para verificarmos qual é a variante aplicada em cada caso).

- Como nos *visitors* o padrão para iterar sobre a árvore sintáctica é explicito (i.e. obriga à invocação do *visit* para o nó da árvore que pretendemos visitar), podemos reutilizar o resultado desses métodos para passar informação entre o nó visitado e o nó visitante.
- Neste caso estamos interessados em passar valores numéricos escalares (`Double`), ou pontos (`Point`), pelo que vamos utilizar o tipo `Object`.
  `antlr4-visitor Shapes InterpreterV Object`
  ou (caso exista apenas uma gramática neste directório):

  `antlr4-visitor InterpreterV Object`

- Podemos também utilizar os resultados dos métodos `visit` para indicar situações de erro (atribuindo esse significado ao resultado `null`).

- Podemos agora começar a implementar o interpretador, primeiro apenas para expressões resultantes de números literais.

```
public class InterpreterV extends ShapesBaseVisitor<Object> {

  @Override
  public Object visitStat(ShapesParser.StatContext ctx) {
    Double res = (Double)visit(ctx.expr());
    if (res != null)
      System.out.println("Result: "+res);
    return res;
  }

  @Override
  public Object visitExprNumber(ShapesParser.ExprNumberContext ctx) {
    return Double.parseDouble(ctx.Number().getText());
  }
}
```

- Para testar este código, precisamos de criar uma classe Java que não só realize as análises sintáctica e léxica do código fonte, como também, caso estas sejam bem sucedidas, itere a árvore sintáctica resultante com o *visitor* criado (`InterpreterV`).

- Vamos assim criar uma classe com um método `main` com todas essas funcionalidades:

`antlr4-main -i -v InterpreterV Shapes main`
ou (caso exista apenas uma gramática e a regra principal for a primeira):

`antlr4-main -i -v InterpreterV`

- Note que podemos criar esta classe com os *visitors* e *listeners* que quisermos (a ordem especificada nos argumentos do comando é mantida). Neste caso apenas temos um *visitor*.
- Podemos agora compilar e executar o interpretador:
  `antlr4-build`
  `echo "3" | antlr4-run`
- Vamos então implementar os restantes métodos do interpretador.

1.2 Exemplo *listener*

- Uma primeira versão (limpa) de um *listener* pode ser gerada com o script `antlr4-listener`
  `antlr4-listener Shapes InterpreterL`
  ou (caso exista apenas uma gramática neste directório):

  `antlr4-listener InterpreterL`

- Notem que são criados *callbacks* separados para cada variante das expressões (de forma similar ao que aconteceu com os *visitors*) evitando, dessa forma, a necessidade de repetir algum *parsing* em `Java`.
- Neste padrão de iteração na árvore sintáctica são criados dois *callbacks* por cada regra sintáctica. Um que é invocado antes de percorrer a subárvore (`enter*`), e outro depois (`exit*`).
- Neste caso só vamos precisar dos métodos `exit*`.

- Esta alternativa de iterar a árvore sintáctica levanta o problema de como passar informação (por exemplo, o valor de uma expressão) entre nós da árvore sintáctica.
- Com os *visitors* podíamos utilizar o resultado dos métodos *visit* para passar essa informação, mas nos *listeners* os métodos são `void`.
- Para resolver este problema vamos colocar atributos na gramática (que funcionam como variáveis no contexto das regras onde são colocados):

```
...
distance returns[Double res]: ...
point returns[Double px, Double py]: ...
expr returns[Double res]: ...
...
```

- Agora podemos adaptar a implementação do *visitor* para o *listener*.

```
public class InterpreterL extends ShapesBaseListener {

   @Override
   public void exitStat(ShapesParser.StatContext ctx) {
      if (ctx.expr().res != null)
         System.out.println("Result: "+ctx.expr().res);
   }

   ...

   @Override
   public void exitPoint(ShapesParser.PointContext ctx) {
      ctx.px = ctx.x.res;
      ctx.py = ctx.y.res;
   }

   @Override
   public void exitExprNumber(ShapesParser.ExprNumberContext ctx) {
      ctx.res = Double.parseDouble(ctx.Number().getText());
   }
}
```

1.3 Exemplo acções na gramática

- A implementação alternativa final assenta na inserção de código na própria gramática.
- Em ANTLR é possível executar código (na linguagem destino) durante a construção da árvore sintáctica.
- Essa execução terá lugar na altura em que essa parte da árvore será construída. Assim, poe exemplo, uma acção no fim de da definição de uma regra sintáctica será executada logo após a geração da parte respectiva da árvore sintáctica.
- Para transmitir informação entre nós da árvore sintáctica utilizam-se atributos.
- Nesta implementação vamos reutilizar os atributos já definidos para a implementação do *listener*.
- Nas acções dentro da gramática, o contexto é acedido com o operador `$`.
- O acesso ao texto de um *token* pode utilizar o método `getText` ou o campo `text`.

- Quando um símbolo aparece várias vezes numa produção, pode-se utilizar *alias* para os distinguir.
- ```
  grammar Shapes;

  main: stat* EOF;
  stat:
      expr {
          if ($expr.res != null)
              System.out.println("Result: "+$expr.res);
      }
    ;
  ...
  point returns[Double px, Double py]:
      '(' x=expr ',' y=expr ')' { // aliases x and y
        $px = $x.res;
        $py = $y.res;
      }
    ;
  ...
  expr returns[Double res]:
    ...
    | Number {
        $res = Double.parseDouble($Number.getText());
      } #ExprNumber
    ;
  ...
  ```
  ```