Tema 1

Compiladores, Linguagens e Gramáticas

Introdução

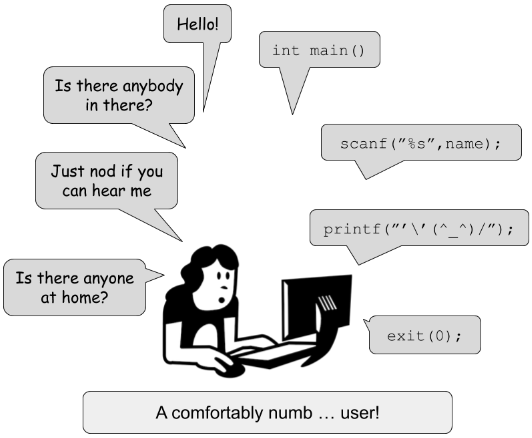
Compiladores, 2º semestre 2021-2022

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

Conteúdo

1 Enquadramento

- Nesta disciplina vamos falar sobre *linguagens* – o que são e como as podemos definir – e sobre *compiladores* – ferramentas que as reconhecem e que permitem realizar acções como consequência desse processo.



- Se tivesse que definir *linguagem* como é que o faria?
  - Podemos dizer que é um protocolo que nos permite *expressar, transmitir e receber ideias*.
  - Até meados do século passado, dir-se-ia que era uma forma de *comunicação* entre pessoas ou, eventualmente, entre seres vivos.
  - Com o advento dos computadores, o conceito de linguagem foi alargado para a comunicação com e entre máquinas.
  - Em comum, a necessidade de ter mais do que uma entidade comunicante, e um código e conjunto de regras que torna essa comunicação inteligível para todas as partes.
  - Uma pessoa a tentar comunicar em português com outra que desconheça essa língua é tão eficaz como tentar pôr um gato a tocar uma pauta de uma sonata de Beethoven (outra linguagem: a música).
- Portanto, é necessário não só ter uma codificação e um conjunto de regras adequadas a cada linguagem, como também interlocutores que as conheçam.
- Curiosamente, línguas diferentes como o português e o inglês, são compostas por *palavras* diferentes, mas partilham muitos dos símbolos utilizados para construir essas palavras.
- Assim, "*adeus*" e "*goodbye*" são sequências diferentes de letras, muito embora tenham um significado semelhante e partilhem também o alfabeto de letras com que são construídas.
- No entanto, dificilmente se reconhece alguma partilha com a tradução para uma das primeiras linguagens escritas – a linguagem cuneiforme babilónica ($\sim$ 3400BC)[1]:

- Por outro lado, a nossa compreensão de um texto não se faz somente lendo a sequência de letras que o compõem, mas sim compreendendo a sua sequência de *ideias*, que são expressas por *frases*, que por sua vez são sequências gramaticalmente correctas de *palavras*, elas sim compostas por sequências de *letras* e outros símbolos do alfabeto.
- Assim, podemos ver uma linguagem natural como o português como sendo composta por mais do que uma linguagem:
  - Uma que explicita as regras para construir palavras a partir do alfabeto das letras:

$$\text{a} + \text{d} + \text{e} + \text{u} + \text{s} \;\rightarrow\; \text{adeus}$$

  - E outra que contém as regras gramaticais para construir frases a partir das palavras resultantes da linguagem anterior:

$$\text{adeus} + \text{e} + \text{até} + \text{amanhã} \;\rightarrow\; \text{adeus e até amanhã}$$

    Neste caso o alfabeto deixa de ser o conjunto de letras e passa a ser o conjunto de palavras válidas.

[1]Traduzido em: https://funtranslations.com/babylonian

- Inerente às linguagens, é a necessidade de decidir se uma sequência de símbolos do alfabeto é válida.
  - **correcto**:
    $\mathrm{a}+\mathrm{d}+\mathrm{e}+\mathrm{u}+\mathrm{s} \rightarrow$ adeus
    adeus $+\,\mathrm{e}+$ até $+$ amanhã $\rightarrow$ adeus e até amanhã
  - **incorrecto**:
    $\mathrm{e}+\mathrm{d}+\mathrm{u}+\mathrm{a}+\mathrm{s} \rightarrow$ edues
    até $+$ adeus $+$ amanhã $+\,\mathrm{e} \rightarrow$ até adeus amanhã e
- Só sequências válidas é que permitem uma comunicação correcta.
- Por outro lado, essa comunicação tem muitas vezes um efeito.
- Seja esse efeito uma resposta à mensagem inicial, ou o despoletar de uma qualquer acção.

1.1 Linguagens de programação

- As linguagens de comunicação com computadores – designadas por linguagens de programação – partilham todas estas características.
- Diferem, no facto de não poderem ter nenhuma *ambiguidade*, e de as acções despoletadas serem muitas vezes a mudança do estado do sistema computacional, podendo este estar ligado a entidades externas como sejam outros computadores, pessoas, sistemas robóticos, máquinas de lavar, etc..
- Vamos ver que podemos definir linguagens de programação por estruturas formais bem comportadas.
- Para além disso, veremos também que essas definições nos ajudam a implementar acções interessantes.

Desenvolvimento das linguagens de programação umbilicalmente ligado com as tecnologias de compilação!

2 Compiladores: Introdução

Compiladores: Compreensão, interpretação e/ou tradução automática de linguagens.

- Os *compiladores* são programas que permitem:
  1. decidir sobre a correcção de sequências de símbolos do respectivo alfabeto;
  2. despoletar acções resultantes dessas decisões.
- Frequentemente, os compiladores "limitam-se" a fazer a tradução entre linguagens.

- É o caso dos compiladores das linguagens de programação de alto nível (`Java`, `C++`, `Eiffel`, etc.), que traduzem o código fonte dessas linguagens em código de linguagens mais próximas do *hardware* do sistema computacional (e.g. *assembly* ou *Java bytecode*).
- Nestes casos, na inexistência de erros, é gerado um programa composto por código executável directa ou indirectamente pelo sistema computacional:

Exemplo: `Java` *bytecode*

```
public class Hello
{
  public static void main(String[] args)
  {
    System.out.println("Hello!");
  }
}
```

```
javac Hello.java
javap -c Hello.class
```

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3  // String Hello!
       5: invokevirtual #4  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

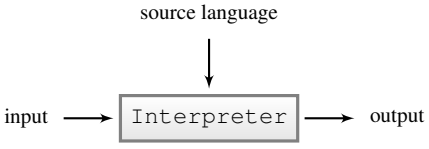
Exemplo 2: Calculadora

- Código fonte:
-
```
1+2*3:4
```

- Uma possível compilação para Java:

```
public class CodeGen {
   public static void main(String[] args) {
      int v2 = 1;
      int v5 = 2;
      int v6 = 3;
      int v4 = v5 * v6;
      int v7 = 4;
      int v3 = v4 / v7;
      int v1 = v2 + v3;
      System.out.println(v1);
   }
}
```

- Uma variante possível consiste num *interpretador*:



- Neste caso a execução é feita instrução a instrução.
- `Python` e `bash` são exemplos de linguagens interpretadas.
- Existem também aproximações hibridas em que existe compilação de código para uma linguagem intermédia, que depois é interpretada na execução.
- A linguagem `Java` utiliza uma estratégia deste género em que o código fonte é compilado para *Java bytecode*, que depois é interpretado pela máquina virtual `Java`.
- Em geral os compiladores processam código fonte em formato de *texto*, havendo uma grande variedade no formato do código gerado (texto, binário, interpretado, . . .).

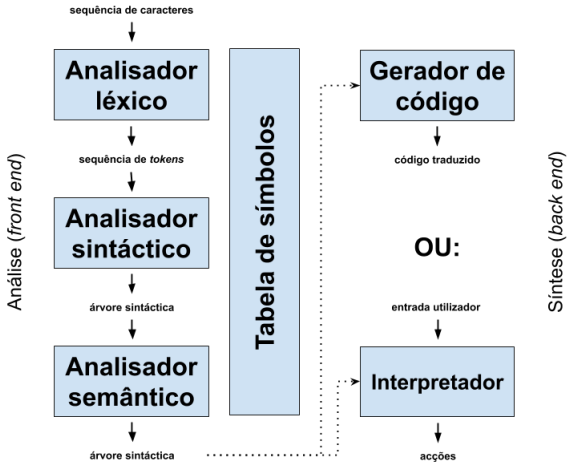
Exemplo: Calculadora

- Código fonte:
-
```
1+2*3:4
```

- Uma possível interpretação:

```
2.5
```

3 Estrutura de um Compilador



- Uma característica interessante da compilação de linguagens de alto nível, é o facto de, tal como no caso das linguagens naturais, essa compilação envolver mais do que uma linguagem:
  - **análise léxica**: composição de letras e outros caracteres em palavras (*tokens*);
  - **análise sintáctica**: composição de *tokens* numa estrutura sintáctica adequada.
  - **análise semântica**: verificação se a estrutura sintáctica tem significado.
- As acções consistem na geração do programa na linguagem destino e podem envolver também diferentes fases de geração de código e optimização.

3.1 Análise Lexical

- Conversão da sequência de caracteres de entrada numa sequência de elementos lexicais.
- Esta estratégia simplifica brutalmente a gramática da análise sintáctica, e permite uma implementação muito eficiente do analisador léxico (mais tarde veremos em detalhe porquê).
- Cada elemento lexical pode ser definido por um tuplo com uma identificação do elemento e o seu valor (o valor pode ser omitido quando não se aplica):

  ```
  <token_name, attribute_value>
  ```

- Exemplo 1:

  ```
  pos = pos + vel * 5;
  ```

  pode ser convertido pelo analisador léxico (*scanner*) em:

  ```
  <id, pos> <=> <id, pos> <+> <id, vel> <*> <int, 5> <;>
  ```

- Em geral os espaços em branco, e as mudanças de linha e os comentários não são relevantes nas linguagens de programação, pelo que podem ser eliminados pelo analisador lexical.

- Exemplo 2: esboço de linguagem de processamento geométrico:

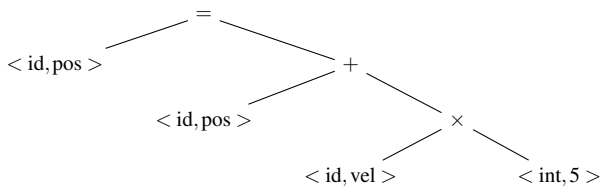
```
distance ( 0 , 0 ) ( 4 , 3 )
```

pode ser convertido pelo analisador léxico (*scanner*) em:

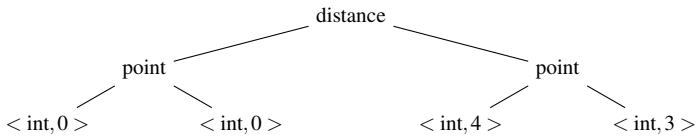
```
<distance> <(> <num,0> <,> <num,0> <)>
<(> <num,4> <,> <num,3> <)>
```

3.2 Análise Sintáctica

- Após a análise lexical segue-se a chamada análise sintáctica (*parsing*), onde se verifica a conformidade da sequência de elementos lexicais com a estrutura sintáctica da linguagem.
- Nas linguagens que se pretende sintacticamente processar, podemos sempre fazer uma aproximação à sua estrutura formal através duma representação tipo *árvore*.
- Para esse fim é necessário uma *gramática* que especifique a estrutura desejada (voltaremos a este problema mais à frente).
- No exemplo 1 (`pos = pos + vel * 5;`):



- No exemplo 2 (`distance ( 0 , 0 ) ( 4 , 3 )`):



- Chama-se a atenção para duas características das árvores sintácticas:
  - não incluem alguns elementos lexicais (que apenas são relevantes para a estrutura formal);
  - definem sem ambiguidade a ordem das operações (havemos de voltar a este problema).

3.3 Análise Semântica

- A parte final do *front end* do compilador é a *análise semântica*.
- Nesta fase são verificadas, tanto quando possível, restrições que não é possível (ou sequer desejável) que sejam feitas nas duas fases anteriores.
- Por exemplo: verificar se um identificador foi declarado, verificar a conformidade no sistema de tipos da linguagem, etc.
- Note-se que apenas restrições com verificação estática (i.e. em tempo de compilação), podem ser objecto de análise semântica pelo compilador.
- Se no exemplo 2 existisse a instrução de um círculo do qual fizesse parte a definição do seu raio, não seria em geral possível, durante a análise semântica, garantir um valor não negativo para esse raio (essa semântica apenas poderia ser verificada dinamicamente, i.e., em tempo de execução).

- Utiliza a árvore sintáctica da análise sintáctica assim como uma estrutura de dados designada por tabela de símbolos (assente em arrays associativos).
- Esta última fase de análise deve garantir o sucesso das fases subsequentes (geração e eventual optimização de código, ou interpretação).

3.4 Síntese

- Havendo garantia de que o código da linguagem fonte é válido, então podemos passar aos efeitos pretendidos com esse código.
- Os efeitos podem ser:
  1. simplesmente a indicação de validade do código fonte;
  2. a tradução do código fonte numa linguagem destino;
  3. ou a interpretação e execução imediata.
- Em todos os casos, pode haver interesse na identificação e localização precisa de eventuais erros.
- Como a maioria do código fonte assenta em texto, é usual indicar não só a instrução mas também a linha onde cada erro ocorre.

Geração de código: exemplo

- No processo de compilação, pode haver o interesse em gerar uma representação intermédia do código que facilite a geração final de código.
- Uma forma possível para essa representação intermédia é o chamado *código de triplo endereço*.
- Para o exemplo 1 (`pos = pos + vel * 5;`) poderíamos ter:

```
t1 = inttofloat(5)
t2 = id(vel) * t1
t3 = id(pos) + t2
id(pos) = t3
```

- Este código poderia depois ser optimizado na fase seguinte da compilação:

```
t1 = id(vel) * 5.0
id(pos) = id(pos) + t1
```

- E por fim, poder-se-ia gerar *assembly* (pseudo-código):

```
LOAD  R2, id(vel)    // load value from memory to register R2
MULT  R2, R2, #5.0   // mult. 5 with R2 and store result in R2
LOAD  R1, id(pos)    // load value from memory to register R1
ADD   R1, R1, R2     // add R1 with R2 and store result in R1
STORE id(pos), R1    // store value to memory from register R1
```

4 Implementação de um Compilador

Para ilustrar o trabalho envolvido em processadores de linguagens vamos implementar "à mão" um interpretador completo para a linguagem sugerida pela instrução: **`distance`** `( 0 , 0 ) ( 4 , 3 ).`

4.1 Análise léxica

Para desenvolvermos "à mão" um analisador léxico sem complicações excessivas, vamos obrigar a que *tokens* da linguagem estejam separados por pelo menos um espaço em branco e/ou uma mudança de linha. Dessa forma, podemos utilizar a classe `Scanner` (métodos `hasNext` e `next`) da biblioteca nativa `Java`.

Como estratégia de base para implementar este analisador, vamos considerar que este tem sempre a si associado um *token* actual. Para o início e fim, existirão dois *tokens* especiais: `NONE` e `EOF`.

Cada *token* tem a si associado o seu tipo, sendo que *tokens* do mesmo tipo partilham as mesmas propriedades léxicas; e, quando aplicável, um atributo (textual) que complete a sua definição.

Este analisador será utilizável por um método (`nextToken`) que vai gerar o próximo *token* consumindo caracteres da entrada.

A listagem 1 mostra uma possível implementação desse programa.

Compilando e executando este programa com a entrada:

```
echo "distance ( 0 , 0 ) ( 1 , 4 )" | java -ea lexer/GeometryLanguageLexer
```

Listing 1: Exemplo de um analisador lexical

```
package lexer;

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

public class GeometryLanguageLexer {
   /**
    * token types
    */
   public enum tokenIds {
      NONE, NUMBER, COMMA, OPEN_PARENTHESES, CLOSE_PARENTHESES, DISTANCE, EOF
   }

   /**
    * Updates actual token to the next input token.
    */
   public static void nextToken() {
      assert token != tokenIds.EOF;

      token = tokenIds.EOF;
      attr = "";
      if (sc.hasNext()) {
         text = sc.next();
         switch(text) {
            case ",": token = tokenIds.COMMA; break;
            case "(": token = tokenIds.OPEN_PARENTHESES; break;
            case ")": token = tokenIds.CLOSE_PARENTHESES; break;
            case "distance": token = tokenIds.DISTANCE; break;
            default:
               attr = text;
               try {
                  value = Double.parseDouble(text);
                  token = tokenIds.NUMBER;
               }
               catch(NumberFormatException e) {
                  err.println("ERROR: unknown lexeme \""+text+"\"");
                  exit(1);
               }
               break;
         }
      }
   }

   /**
    * Actual token type
    */
   public static tokenIds token() { return token; }
   /**
    * Actual token attribute
    */
   public static String attr() { return attr; }
   /**
    * Actual token value
    */
   public static Double value() { return value; }

   public static void main(String[] args) {
      do {
         nextToken();
         out.print(" <"+token()+(attr().length() > 0 ? ","+attr() : "")+">");
      }
      while(token() != tokenIds.EOF);
      out.println();
   }

   protected static final Scanner sc = new Scanner(System.in);

   protected static tokenIds token = tokenIds.NONE;
   protected static String text = "";
   protected static String attr;
   protected static double value;

}
```

obtemos a seguinte saída:

<OP_DISTANCE> <OPEN_PARENTHESES> <NUMBER,0> <COMMA> <NUMBER,0> <CLOSE_PARENTHESES>
<OPEN_PARENTHESES> <NUMBER,1> <COMMA> <NUMBER,4> <CLOSE_PARENTHESES> <EOF>

4.2 Análise sintáctica

Como primeira aproximação para a construção de uma analisador sintáctico vamos fazer com que este apenas indique se o código fonte é uma sequência válida de *tokens* (ou não). Com esse objectivo, vamos seguir a seguinte estratégia:

- Identificar as estruturas importantes da linguagem (regras);
- Associar métodos booleanos ao reconhecimento de cada regra;
- Garantir que, na invocação desses métodos, o *token* actual é o que seria de esperar no inicio dessas regras;
- O processo de reconhecimento de regras terá três comportamentos possíveis:
  1. Se for bem sucedido, todos os tokens associados à regra foram consumidos (i.e. fazem parte do passado do analisador léxico);
  2. Falha por não reconhecimento do primeiro *token* da regra. Neste caso não há lugar ao consumo de nenhum token;
  3. Falha no meio do reconhecimento da regra. Nesta situação, o analisador limita-se a rejeitar a sequência de *tokens*.
- Neste processo, sempre que é reconhecido um *token*, o analisador sintáctico consumirá esse *token* (i.e. avança para o próximo).

As estruturas (regras) importantes no esboço apresentado são a instrução *distância*. Como esta instrução se aplica a dois pontos, temos também a regra *ponto* (que por sua vez se aplica a um par de números).

A listagem 2 mostra uma possível implementação desse programa.

4.3 Análise semântica

A linguagem definida até agora não permite erros semânticos que possam servir de exemplo para esta secção. Para resolver esse problema, vamos acrescentar à linguagem a possibilidade de definir e utilizar *variáveis*. A existência de variáveis possibilita a existência de erros semânticos resultantes da utilização de variáveis não definidas.

As variáveis, são um recurso programático que permite o armazenamento de valores recorrendo a nomes (designados *identificadores*). Nesta linguagem, vamos definir um identificador como sendo uma sequência não vazia de letras minúsculas (sem acentos).

O analisador léxico tem de ser alterado, acrescentando os *tokens* `ID` e `EQUAL`:

```
...
    public enum tokenIds {
        NONE, NUMBER, COMMA, OPEN_PARENTHESES, CLOSE_PARENTHESES, DISTANCE, ID, EQUAL, EOF
    }
    public static void nextToken() {
...
                case "=": token = tokenIds.EQUAL; break;
                default:
                    attr = text;
                    if (attr.matches("[a-z]+"))
                        token = tokenIds.ID;
                    else
                    {
                        try {
                            value = Double.parseDouble(text);
                            token = tokenIds.NUMBER;
                        }
                        catch(NumberFormatException e) {
                            err.println("ERROR: unknown lexeme \""+text+"\"");
```

Listing 2: Exemplo de um analisador sintáctico

```
package parser;

import static java.lang.System.*;
import static lexer.GeometryLanguageLexer.*;

public class GeometryLanguageParser {
   /**
    * Start rule: attempts to parse the whole input.
    */
   public static boolean parse() {
      assert token() == tokenIds.NONE;

      nextToken();
      return parseDistance();
   }

   /**
    * Distance rule parsing.
    */
   public static boolean parseDistance() {
      assert token() != tokenIds.NONE;

      boolean result = token() == tokenIds.DISTANCE;
      if (result) {
         nextToken();
         result = parsePoint();
         if (result) {
            result = parsePoint();
         }
      }
      return result;
   }

   /**
    * Point rule parsing.
    */
   public static boolean parsePoint() {
      assert token() != tokenIds.NONE;

      boolean result = token() == tokenIds.OPEN_PARENTHESES;
      if (result) {
         nextToken();
         result = token() == tokenIds.NUMBER;
         if (result) {
            nextToken();
            result = token() == tokenIds.COMMA;
            if (result) {
               nextToken();
               result = token() == tokenIds.NUMBER;
               if (result) {
                  nextToken();
                  result = token() == tokenIds.CLOSE_PARENTHESES;
                  if (result) {
                     nextToken();
                  }
               }
            }
         }
      }
      return result;
   }

   public static void main(String[] args) {
      if(parse())
         out.println("Ok");
      else
         out.println("ERROR");
   }

}
```

Listing 3: Expressões.

```
    public static boolean parseExpression() {
        assert token() != tokenIds.NONE;

        boolean result = true;
        switch(token()) {
            case NUMBER:
                nextToken();
                break;
            case ID:
                if (!symbolTable.containsKey(attr()))
                {
                    err.println("ERROR: undefined variable \""+attr()+"\"");
                    exit(1);
                }
                nextToken();
                break;
            default:
                result = parseDistance();
                break;
        }
        return result;
    }
```

```
                        exit(1);
                    }
                }
                break;
...
    }
...
```

A estrutura de dados adequada para lidar com variáveis é o *array* associativo. Com esta estrutura de dados, podemos associar ao nome da variável os valores que quisermos. Para já apenas queremos saber se a variável está, ou não está, definida. Pelo que basta a existência do identificador no *array* associativo.

```
    protected static Map<String,Object> symbolTable = new HashMap<String,Object>();
```

Vamos também generalizar um pouco a linguagem definido o conceito de *expressão*. Uma expressão vai ser uma entidade do programa que tem a si associada um valor numérico. No caso, poderá ser um número literal, uma variável ou a instrução de distância. O código 3 apresenta um método que faz essa análise sintáctica.

Para activar esta generalização, basta substituir a análise sintáctica de número literal (`NUMBER`) por uma invocação deste novo método. Note que esta nova estrutura sintáctica aumenta imenso a flexibilidade da linguagem, já que agora onde se espera um valor numérico (coordenada de um ponto, atribuição de valor) pode aparecer uma qualquer expressão (em vez de somente um número literal).

Precisamos agora de acrescentar uma instrução de atribuição de valor (que define, ou redefine, o valor duma variável). O código 4 mostra esse método.

Para tornar activa a nova instrução vamos modificar o método inicial de análise sintáctica (código 5).

4.4 Síntese: interpretação do código

Para completar este exemplo, falta apenas implementar acções ligadas à linguagem definida. Para não complicar o problema, vamos considerar que todas as instruções têm um valor numérico, e que o efeito de uma instruções é a escrita desse valor. Assim, por exemplo, a aplicação da instrução de distância deve calcular e escrever esse valor.

Para implementar este comportamento vamos inserir directamente no analisador sintáctico o código necessário para realizar estas acções. Como as instruções passam a estar associadas a valores numéricos, precisamos de arranjar forma de lhes associar esses valores. Nesse sentido, vamos substituir os resultados booleanos pelo tipo de dados não primitivo `Double`. Um resultado igual a `null` indica erro sintáctico, e

Listing 4: Atribuição de valor.

```
public static boolean parseAssignment() {
   assert token() != tokenIds.NONE;

   boolean result = token() == tokenIds.ID;
   if (result) {
      String var = attr();
      nextToken();
      result = token() == tokenIds.EQUAL;
      if (result) {
         nextToken();
         result = parseExpression();
         if (result) {
            symbolTable.put(var, null);
         }
      }
   }
   return result;
}
```

Listing 5: Novo método *parse*.

```
public static boolean parse() {
   assert token() == tokenIds.NONE;

   nextToken();
   boolean result = true;
   while(result && token() != tokenIds.EOF) {
      result = parseDistance();
      if (!result)
         result = parseAssignment();
   }
   return result;
}
```

um valor não nulo, expressa o valor associado à instrução. A única excepção a este procedimento será a análise sintáctica de pontos já que estes têm de estar associados a um par de valores (e não apenas a um).

O código 6 exemplifica o código do interpretador (em anexo é fornecido o programa completo).

5 Linguagens: Definição como Conjunto

- As linguagens servem para *comunicar*.
- Uma mensagem pode ser vista como uma sequência de *símbolos*.
- No entanto, uma linguagem não aceita todo o tipo de símbolos e de sequências.
- Uma linguagem é caracterizada por um conjunto de símbolos e uma forma de descrever sequências válidas desses símbolos (i.e. o conjunto de sequências válidas).
- Se as linguagens naturais admitem alguma subjectividade e ambiguidade, as linguagens de programação requerem total objectividade.

- Como definir linguagens de forma sintética e objectiva?
- Definir por *extensão* – isto é, enumerando todas as possíveis ocorrências – é uma possibilidade.
- No entanto, para linguagens minimamente interessantes não só teríamos uma descrição gigantesca como também, provavelmente, incompleta.
- As linguagens de programação tendem a aceitar variantes infinitas de entradas.
- Alternativamente podemos descrevê-la por *compreensão*.
- Uma possibilidade é utilizar os formalismos ligados à definição de *conjuntos*.

5.1 Conceito básicos e terminologia

- Um conjunto pode ser definido por *extensão* (ou enumeração) ou por *compreensão*.
- Um exemplo de um conjunto definido por extensão é o conjunto dos algarismos binários $\{0,1\}$.
- Na definição por compreensão utiliza-se a seguinte notação:

$$\{x \mid p(x)\}$$

ou

$$\{x : p(x)\}$$

- x é a variável que representa um qualquer elemento do conjunto, e $p(x)$ um predicado sobre essa variável.
- Assim, este conjunto é definido contendo todos os valores de x em que o predicado $p(x)$ é verdadeiro.
- Por exemplo: $\{n \mid n \in \mathbb{N} \wedge n \leq 9\} \quad = \quad \{1,2,3,4,5,6,7,8,9\}$

- Um *símbolo* (ou *letra*) é a unidade atómica (indivisível) das linguagens.
- Em linguagens assentes em texto, um símbolo será um carácter.
- Um *alfabeto* é um conjunto finito não vazio de símbolos.
- Por exemplo:
  - $A = \{0,1\}$ é o alfabeto dos algarismos binários.
  - $A = \{0,1,\cdots,9\}$ é o alfabeto dos algarismos decimais.
- Uma *palavra* (*string* ou cadeia) é uma sequência de símbolos sobre um dado alfabeto A.

$$U = a_1 a_2 \cdots a_n, \quad \text{com} \quad a_i \in A \wedge n \geq 0$$

- Por exemplo:
  - $A = \{0,1\}$ é o alfabeto dos algarismos binários.

    $01101, 11, 0$
  - $A = \{0,1,\cdots,9\}$ é o alfabeto dos algarismos decimais.

Listing 6: Interpretador.

```
...
    public static Double parseAssignment() {
        assert token() != tokenIds.NONE;

        Double result = null;
        if (token() == tokenIds.ID) {
            String var = attr();
            nextToken();
            if (token() == tokenIds.EQUAL) {
                nextToken();
                result = parseExpression();
                if (result != null) {
                    symbolTable.put(var, result);
                }
            }
        }
        return result;
    }
...
    public static Double parseDistance() {
        assert token() != tokenIds.NONE;

        Double result = null;
        if (token() == tokenIds.DISTANCE) {
            nextToken();
            Double[] p1 = parsePoint();
            if (p1 != null) {
                Double[] p2 = parsePoint();
                if (p2 != null) {
                    result = Math.sqrt(Math.pow(p1[0]-p2[0],2)+Math.pow(p1[1]-p2[1],2));
                }
            }
        }
        return result;
    }
...
    public static Double[] parsePoint() {
        assert token() != tokenIds.NONE;

        Double[] result = null;
        if (token() == tokenIds.OPEN_PARENTHESES) {
            nextToken();
            Double x = parseExpression();
            if (x != null) {
                if (token() == tokenIds.COMMA) {
                    nextToken();
                    Double y = parseExpression();
                    if (y != null) {
                        if (token() == tokenIds.CLOSE_PARENTHESES) {
                            nextToken();
                            result = new Double[2];
                            result[0] = x;
                            result[1] = y;
                        }
                    }
                }
            }
        }
        return result;
    }
...
```

$2016, 234523, 999999999999999, 0$

- $A = \{0, 1, \cdots, 0, a, b, \cdots, z, @, \cdots\}$

  mos@ua.pt, Bom dia!

- A *palavra vazia* é uma sequência de zero símbolos e denota-se por ε (épsilon).
- Note que ε não pertence ao alfabeto.
- Uma *sub-palavra* de uma palavra u é uma sequência contígua de 0 ou mais símbolos de u.
- Um *prefixo* de uma palavra u é uma sequência contígua de 0 ou mais símbolos iniciais de u.
- Um *sufixo* de uma palavra u é uma sequência contígua de 0 ou mais símbolos terminais de u.
- Por exemplo:
  - as é uma sub-palavra de casa, mas não prefixo nem sufixo
  - 001 é prefixo e sub-palavra de 00100111 mas não é sufixo
  - ε é prefixo, sufixo e sub-palavra de qualquer palavra u
  - qualquer palavra u é prefixo, sufixo e sub-palavra de si própria
- O *fecho* (ou conjunto de cadeias) do alfabeto A denominado por A^*, representa o conjunto de todas as palavras definíveis sobre o alfabeto A, incluindo a palavra vazia.
- Por exemplo:
  - $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \cdots\}$
  - $\{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}^* = \{\varepsilon, \clubsuit, \diamondsuit, \heartsuit, \spadesuit, \clubsuit\diamondsuit, \cdots\}$
- Dado um alfabeto A, uma *linguagem* L sobre A é um conjunto finito ou infinito de palavras consideradas válidas definidas com símbolos de A.
  Isto é: $L \subseteq A^*$
- Exemplo de linguagens sobre o alfabeto $A = \{0,1\}$
  - $L_1 = \{u \mid u \in A^* \wedge |u| \leq 2\} = \{\varepsilon, 0, 1, 00, 01, 10, 11\}$
  - $L_2 = \{u \mid u \in A^* \wedge \forall_i u_i = 0\} = \{\varepsilon, 0, 00, 000, 0000, \cdots\}$
  - $L_3 = \{u \mid u \in A^* \wedge u.\text{count}(1)\ mod\ 2 = 0\} = \{000, 11, 000110101, \cdots\}$
  - $L_4 = \{\} = \emptyset$ (conjunto vazio)
  - $L_5 = \{\varepsilon\}$
  - $L_6 = A$
  - $L_7 = A^*$
- Note que $\{\}$, $\{\varepsilon\}$, A e A^* são linguagens sobre o alfabeto A qualquer que seja A
- Uma vez que as linguagens são conjuntos, todas as operações matemáticas sobre conjuntos são aplicáveis: reunião, intercepção, complemento, diferença, etc.

5.2 Operações sobre palavras

- O *comprimento* de uma palavra u denota-se por $|u|$ e representa o seu número de símbolos.
- O comprimento da palavra vazia é zero

$$|\varepsilon| = 0$$

- É habitual interpretar-se a palavra u como uma função de acesso aos seus símbolos (tipo *array*):

$$u : \{1, 2, \cdots, n\} \rightarrow A, \quad \text{com} \quad n = |u|$$

em que u_i representa o $i$ésimo símbolo de u

- O *reverso* de uma palavra u é a palavra, denota-se por u^R, e é obtida invertendo a ordem dos símbolos de u

$$u = \{u_1, u_2, \cdots, u_n\} \implies u^R = \{u_n, \cdots, u_2, u_1\}$$

- A *concatenação* (ou *produto*) das palavras u e v denota-se por $u.v$, ou simplesmente uv, e representa a justaposição de u e v, i.e., a palavra constituída pelos símbolos de u seguidos pelos símbolos de v.
- Propriedades da concatenação:
  - $|u.v| = |u| + |v|$
  - $u.(v.w) = (u.v).w = u.v.w$ (associatividade)
  - $u.\varepsilon = \varepsilon.u = u$ (elemento neutro)
  - $u \neq \varepsilon \wedge v \neq \varepsilon \wedge u \neq v \implies u.v \neq v.u$ (não comutativo)
- A *potência* de ordem n, com $n \geq 0$, de uma palavra u denota-se por u^n e representa a concatenação de n réplicas de u, ou seja, $\underbrace{uu\cdots u}_{n\times}$.
- $u^0 = \varepsilon$

5.3 Operações sobre linguagens

Operações sobre linguagens: reunião

- A *reunião* de duas linguagens L_1 e L_2 denota-se por $L_1 \cup L_2$ e é dada por:

$$L_1 \cup L_2 = \{u \mid u \in L_1 \vee u \in L_2\}$$

- Por exemplo, se definirmos as linguagens L_1 e L_2 sobre o alfabeto $A = \{a, b\}$:

$$\begin{aligned} L_1 &= \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\} \\ L_2 &= \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\} \end{aligned}$$

- qual será o resultado da reunião destas linguagens?

$$L = L_1 \cup L_2 = ?$$

- Resposta:

$$L = \{w_1 a w_2 \mid w_1, w_2 \in A^* \wedge (w_1 = \varepsilon \vee w_2 = \varepsilon)\}$$

Operações sobre linguagens: intercepção

- A *intercepção* de duas linguagens L_1 e L_2 denota-se por $L_1 \cap L_2$ e é dada por:

$$L_1 \cap L_2 = \{u \mid u \in L_1 \wedge u \in L_2\}$$

- Por exemplo, se definirmos as linguagens L_1 e L_2 sobre o alfabeto $A = \{a, b\}$:

$$\begin{aligned} L_1 &= \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\} \\ L_2 &= \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\} \end{aligned}$$

- qual será o resultado da intercepção destas linguagens?

$$L = L_1 \cap L_2 = ?$$

- Resposta:

$$L = \{awa \mid w \in A^*\} \cup \{a\}$$

Operações sobre linguagens: diferença

- A *diferença* de duas linguagens L_1 e L_2 denota-se por $L_1 - L_2$ e é dada por:

$$L_1 - L_2 = \{u \mid u \in L_1 \wedge u \notin L_2\}$$

- Por exemplo, se definirmos as linguagens L_1 e L_2 sobre o alfabeto $A = \{a, b\}$:

$$L_1 = \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\}$$
$$L_2 = \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\}$$

- qual será o resultado da diferença destas linguagens?

$$L = L_1 - L_2 = ?$$

- Resposta:

$$L = \{awx \mid w \in A^* \wedge x \in A \wedge x \neq a\}$$

- ou:

$$L = \{awb \mid w \in A^*\}$$

Operações sobre linguagens: complementação

- A *complementação* da linguagem L denota-se por $\overline{L}$ e é dada por:

$$\overline{L} = A^* - L = \{u \mid u \notin L\}$$

- Por exemplo, se definirmos a linguagem L_1 sobre o alfabeto $A = \{a, b\}$:

$$L_1 = \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\}$$

- qual será o resultado da complementação desta linguagem?

$$L = \overline{L_1} = ?$$

- Resposta:

$$L = \{xw \mid w \in A^* \wedge x \in A \wedge x \neq a\} \cup \{\varepsilon\}$$

- ou:

$$L = \{bw \mid w \in A^*\} \cup \{\varepsilon\}$$

Operações sobre linguagens: concatenação

- A *concatenação* de duas linguagens L_1 e L_2 denota-se por $L_1.L_2$ e é dada por:

$$L_1.L_2 = \{uv \mid u \in L_1 \wedge v \in L_2\}$$

- Por exemplo, se definirmos as linguagens L_1 e L_2 sobre o alfabeto $A = \{a,b\}$:

$$L_1 = \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\}$$
$$L_2 = \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\}$$

- qual será o resultado da concatenação destas linguagens?

$$L = L_1.L_2 = ?$$

- Resposta:

$$L = \{awa \mid w \in A^*\}$$

Operações sobre linguagens: potenciação

- A *potência* de ordem n da linguagem L denota-se por L^n e é definida indutivamente por:

$$L^0 = \{\varepsilon\}$$
$$L^{n+1} = L^n.L$$

- Por exemplo, se definirmos a linguagem L_1 sobre o alfabeto $A = \{a,b\}$:

$$L_1 = \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\}$$

- qual será o resultado da potência de ordem 2 desta linguagem?

$$L = L_1^2 = ?$$

- Resposta:

$$L = \{aw_1aw_2 \mid w_1, w_2 \in A^*\}$$

Operações sobre linguagens: fecho de Kleene

- O *fecho de Kleene* da linguagem L denota-se por L^* e é dado por:

$$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots = \bigcup_{i=0}^{\infty} L^i$$

- Por exemplo, se definirmos a linguagem L_1 sobre o alfabeto $A = \{a,b\}$:

$$L_1 = \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\}$$

- qual será o fecho de Kleene desta linguagem?

$$L = L_1^* = ?$$

- Resposta:

$$L = L_1 \cup \{\varepsilon\}$$

- Note que para $n > 1 \quad L_1^n \subset L_1$

Operações sobre linguagens: notas adicionais

- Note que nas operações binárias sobre conjuntos não é requerido que as duas linguagens estejam definidos sobre o mesmo alfabeto.
- Assim se tivermos duas linguagens L_1 e L_2 definidas respectivamente sobre os alfebetos A_1 e A_2, então o alfabeto resultante da aplicação duma qualquer operação binária sobre as linguagens é: $A_1 \cup A_2$

6 Introdução às gramáticas

- A utilização de conjuntos para definir linguagens não é frequentemente a forma mais adequada e versátil para as descrever.
- Muitas vezes é preferível identificar estruturas intermédias, que abstraem partes ou subconjuntos importantes, da linguagem.
- Tal como em programação, muitas vezes descrições recursivas são bem mais simples, sem perda da objectividade e do rigor necessários.
- É nesse caminho que encontramos as *gramáticas*.
- As *gramáticas* descrevem linguagens por compreensão recorrendo a representações *formais* e (muitas vezes) *recursivas*.
- Vendo as linguagens como sequências de símbolos (ou palavras), as gramáticas definem formalmente as sequências *válidas*.

- Por exemplo, em português a frase "O cão ladra" pode ser gramaticalmente descrita por:

$$\begin{aligned}
\text{frase} &\rightarrow \text{sujeito predicado} \\
\text{sujeito} &\rightarrow \text{artigo substantivo} \\
\text{predicado} &\rightarrow \text{verbo} \\
\text{artigo} &\rightarrow \textbf{O} \mid \textbf{Um} \\
\text{substantivo} &\rightarrow \textbf{cão} \mid \textbf{lobo} \\
\text{verbo} &\rightarrow \textbf{ladra} \mid \textbf{uiva}
\end{aligned}$$

- Esta gramática (não recursiva) descreve formalmente 8 possíveis frases, o que é ainda pouco interessante.
- No entanto, contém mais informação do que a frase original, já que classifica os vários elementos da frase (sujeito, predicado, etc.).
- Contém 6 *símbolos terminais* e 6 *símbolos não terminais*.
- Um símbolo não terminal é definido por uma *produção* descrevendo possíveis representações desse símbolo, em função de símbolos terminais e/ou não terminais.

- Formalmente, uma gramática é um quádruplo $G = (T, N, S, P)$, onde:

  1. T é um conjunto finito não vazio designado por alfabeto terminal, onde cada elemento é designado por símbolo *terminal*;
  2. N é um conjunto finito não vazio, disjunto de T ($N \cap T = \emptyset$), cujos elementos são designados por símbolos *não terminais*;
  3. $S \in N$ é um símbolo não terminal específico designado por *símbolo inicial*;
  4. P é um conjunto finito de *regras* (ou produções) da forma $\alpha \rightarrow \beta$ onde $\alpha \in (T \cup N)^* N (T \cup N)^*$ e $\beta \in (T \cup N)^*$, isto é, α é uma cadeia de símbolos terminais e não terminais contendo, pelo menos, um símbolo não terminal; e β é uma cadeia de símbolos, eventualmente vazia, terminais e não terminais.

Gramáticas: exemplos

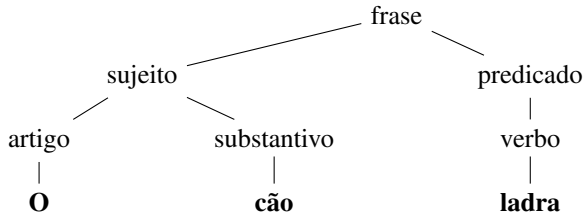
- Formalmente, a gramática anterior será:

$$\begin{aligned} G \;=\; & (\{\mathbf{O}, \mathbf{Um}, \mathbf{cão}, \mathbf{lobo}, \mathbf{ladra}, \mathbf{uiva}\}, \\ & \{\text{frase}, \text{sujeito}, \text{predicado}, \text{artigo}, \text{substantivo}, \text{verbo}\}, \\ & \text{frase}, P) \end{aligned}$$

- P é constituído pelas regras já apresentadas:

$$\begin{aligned} \text{frase} &\rightarrow \text{sujeito predicado} \\ \text{sujeito} &\rightarrow \text{artigo substantivo} \\ \text{predicado} &\rightarrow \text{verbo} \\ \text{artigo} &\rightarrow \mathbf{O} \mid \mathbf{Um} \\ \text{substantivo} &\rightarrow \mathbf{cão} \mid \mathbf{lobo} \\ \text{verbo} &\rightarrow \mathbf{ladra} \mid \mathbf{uiva} \end{aligned}$$

- Podemos descrever a frase "O cão ladra" com a seguinte árvore (denominada sintáctica).



- Considere a seguinte gramática $G = (\{0,1\}, \{S,A\}, S, P)$, onde P é constituído pelas regras:

$$\begin{aligned} S &\rightarrow 0S \\ S &\rightarrow 0A \\ A &\rightarrow 0A1 \\ A &\rightarrow \varepsilon \end{aligned}$$

- Qual será a linguagem definida por esta gramática?

$$L = \{0^n 1^m : n \in \mathbb{N} \wedge m \in \mathbb{N}_0 \wedge n > m\}$$

- Sendo $A = \{a,b\}$, defina uma gramática para a seguinte linguagem:

$$L_1 \;=\; \{aw \mid w \in A^*\}$$

- A gramática $G = (\{a,b\}, \{S,X\}, S, P)$, onde P é constituído pelas regras:

$$\begin{aligned} S &\rightarrow aX \\ X &\rightarrow aX \\ X &\rightarrow bX \\ X &\rightarrow \varepsilon \end{aligned}$$

ou:

$$S \rightarrow aX$$
$$X \rightarrow aX \mid bX \mid \varepsilon$$

- Sendo $A = \{0,1\}$, defina uma gramática para a seguinte linguagem:

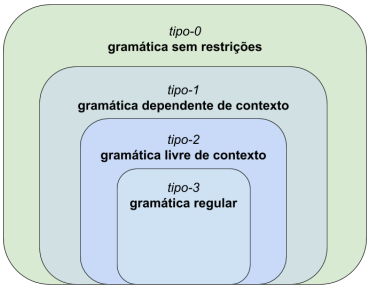
$$L_3 = \{u \mid u \in A^* \wedge u.\text{count}(1)\ mod\ 2 = 0\}$$

- A gramática $G = (\{0,1\},\{S,A\},S,P)$, onde P é constituído pelas regras:

$$S \rightarrow S\,1\,S\,1\,S \mid A$$
$$A \rightarrow 0\,A \mid \varepsilon$$

6.1 Hierarquia de Chomsky

- Restrições sobre α e β permitem definir uma taxonomia das linguagens – hierarquia de Chomsky:
  1. Se não houver nenhuma restrição, G é designada por gramática do *tipo*-0.
  2. G será do *tipo*-1, ou gramática *dependente do contexto*, se cada regra $\alpha \rightarrow \beta$ de P obedece a $|\alpha| \leq |\beta|$ (com a excepção de também poder existir a produção vazia: $S \rightarrow \varepsilon$).
  3. G será do *tipo*-2, ou gramática *independente, ou livre, do contexto*, se cada regra $\alpha \rightarrow \beta$ de P obedece a $|\alpha| = 1$, isto é: α é constituído por um só não terminal.
  4. G será do *tipo*-3, ou gramática *regular*, se cada regra tiver uma das formas: $A \rightarrow cB, A \rightarrow c$ ou $A \rightarrow \varepsilon$, onde A e B são símbolos não terminais (A pode ser igual a B) e c um símbolo terminal. Isto é, em todas as produções, o β só pode ter no máximo um símbolo não terminal sempre à direita (ou, alternativamente, sempre à esquerda).

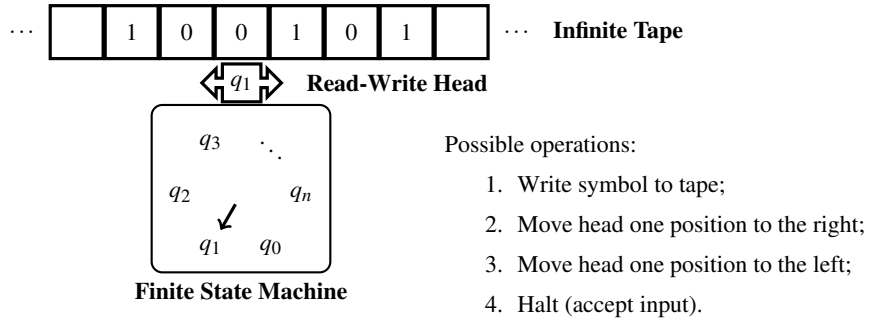


- Para cada um desses tipos podem ser definidos diferentes tipos de máquinas (algoritmos, autómatos) que as podem reconhecer.
- Quanto mais simples for a gramática, mas simples e eficiente é a máquina que reconhece essas linguagens.
- Cada classe de linguagens do *tipo-i* contém a classe de linguagens *tipo*-$(i+1)$ $(i = 0,1,2)$
- Esta hierarquia não traduz apenas as características formais das linguagens, mas também expressam os requisitos de computação necessários:
  1. As *máquinas de Turing* processam gramáticas sem restrições (tipo-0);
  2. Os *autómatos linearmente limitados* processam gramáticas dependentes do contexto (tipo-1);
  3. Os *autómatos de pilha* processam gramáticas independentes do contexto (tipo-2);
  4. Os *autómatos finitos* processam gramáticas regulares (tipo-3).

6.2 Autómatos

6.2.1 Máquina de Turing

- (Alan Turing, 1936)
- Modelo abstracto de computação.
- Permite (em teoria) implementar qualquer programa computável.
- Assenta numa máquina de estados finita, numa "cabeça"de leitura/escrita de símbolos e numa fita infinita (onde se escreve ou lê esses símbolos).
- A "cabeça"de leitura/escrita pode movimentar-se uma posição para esquerda ou direita.
- Modelo muito importante na teoria da computação.
- Pouco relevante na implementação prática de processadores de linguagens.



- A máquina de estados finita (FSM) tem acesso ao símbolo actual e decide a próxima acção a ser realizada.
- A acção consiste na transição de estado e qual a operação sobre a fita.
- Se não for possível nenhuma acção, a entrada é rejeitada.

Máquina de Turing: exemplo

- Dado o alfabeto $A = \{0,1\}$, e considerando que um número inteiro não negativo n é representado pela sequência de $n+1$ símbolos 1, vamos implementar uma MT que some os próximos (i.e à direita da posição actual) dois números inteiros existentes na fita (separados apenas por um 0).
- O algoritmo pode ser simplemente trocar o símbolo 0 entre os dois números por 1, e trocar os dois últimos símbolos 1 por 0.
- Por exemplo: $3+2$ a que corresponde o seguinte estado na fita (símbolo a negrito é a posição da "cabeça"): $\cdots\mathbf{0}111101110\cdots$ (o resultado pretendido será: $\cdots\mathbf{0}111111000\cdots$).
- Considerando que os estados são designados por $E_i, i \geq 1$ (sendo E_1 o estado inicial); e as operações:

  d mover uma posição para a direita;

  e mover uma posição para a esquerda;

  0 escrever o símbolo 0 na fita;

  1 escrever o símbolo 1 na fita;

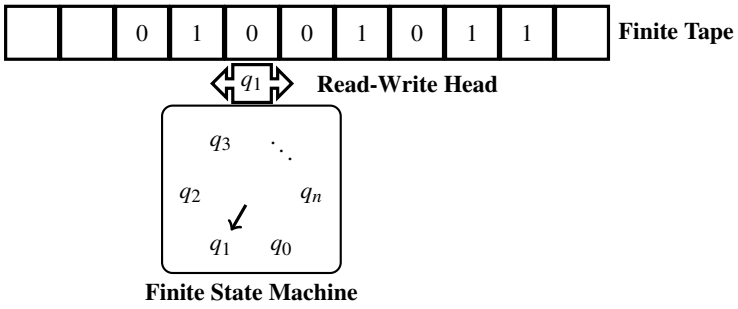
  h aceitar e terminar autómato.

- Uma solução possível é dada pela seguinte diagrama de transição de estados:

	Entrada	
Estado	**0**	**1**
E_1	E_1/d	E_2/d
E_2	$E_3/1$	E_2/d
E_3	E_4/e	E_3/d
E_4	$--$	$E_5/0$
E_5	E_5/e	$E_6/0$
E_6	E_7/e	$--$
E_7	E_1/h	E_7/e

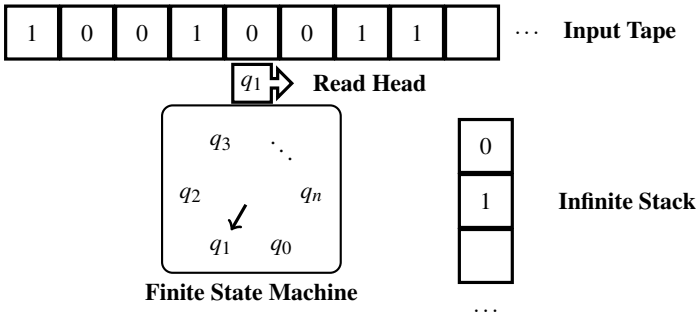
- $E_1\cdots\mathbf{0}111101110\cdots \rightarrow E_1\cdots0\mathbf{1}11101110\cdots \overset{*}{\rightarrow} E_2\cdots01111\mathbf{0}1110\cdots \rightarrow E_3\cdots0111\mathbf{1}1110\cdots \rightarrow E_3\cdots011111\mathbf{1}110\cdots \overset{*}{\rightarrow} E_3\cdots0111111\mathbf{1}0\cdots \rightarrow E_4\cdots0111111\mathbf{1}0\cdots \rightarrow E_5\cdots0111111\mathbf{0}0\cdots \rightarrow E_5\cdots0111111\mathbf{1}00\cdots \rightarrow E_6\cdots0111111\mathbf{0}00\cdots \rightarrow E_7\cdots011111\mathbf{1}000\cdots \overset{*}{\rightarrow} E_7\cdots\mathbf{0}111111000\cdots$

6.2.2 Autómatos linearmente limitados



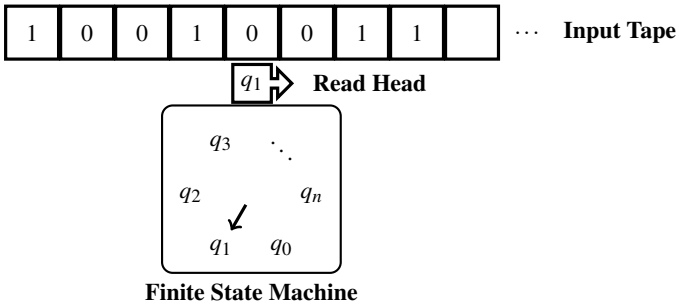
- Diferem das MT pela finitude da fita.

6.2.3 Autómatos de pilha



- "Cabeça"apenas de leitura e suporte de uma pilha sem limites.
- Movimento da "cabeça"apenas numa direcção.
- Autómatos adequados para análise sintáctica.

6.2.4 Autómatos finitos



- Sem escrita de apoio à máquina de estados.
- Autómatos adequados para análise léxica.

Tema 2

ANTLR4

Introdução, Estrutura, Aplicação

Compiladores, 2º semestre 2021-2022

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

Conteúdo

1 Apresentação

- *ANother Tool for Language Recognition*
- O ANTLR é um gerador de processadores de linguagens que pode ser utilizado para ler, processar, executar ou traduzir linguagens.
- Desenvolvido por Terrence Parr:

1988: tese de mestrado (YUCC)

1990: PCCTS (ANTLR v1). Programado em C++.

1992: PCCTS v 1.06

1994: PCCTS v 1.21 e SORCERER

1997: ANTLR v2. Programado em Java.

2007: ANTLR v3 (LL(*), *auto-backtracking*, yuk!).

2012: ANTLR v4 (ALL(*), *adaptive LL*, yep!).

- Terrence Parr, The Definitive ANTLR 4 Reference, 2012, The Pragmatic Programmers.
- Terrence Parr, Language Implementation Patterns, 2010, The Pragmatic Programmers.
- https://www.antlr.org

ANTLR4: instalação

- Descarregar o ficheiro antlr4-install.zip do *elearning*.
- Executar o *script* ./install.sh no directório antlr4-install.
- Há dois ficheiros jar importantes:
  antlr-4.*-complete.jar e antlr-runtime-4.*.jar
- O primeiro é *necessário* para *gerar* processadores de linguagens, e o segundo é o *suficiente* para os *executar*.
- Para experimentar basta:
  ```
  java -jar antlr-4.*-complete.jar
  ```
  ou:
  ```
  java -cp .:antlr-4.*-complete.jar org.antlr.v4.Tool
  ```
- O ANTLR4 fornece uma ferramenta de teste muito flexível (implementada com o script antlr4-test):
  ```
  java org.antlr.v4.gui.TestRig
  ```
- Podemos executar uma gramática sobre uma qualquer entrada, e obter a lista de *tokens* gerados, a árvore sintáctica (num formato tipo LISP), ou mostrar graficamente a árvore sintáctica.

- Nesta disciplina são disponibilizados vários comandos (em bash) para simplificar (ainda mais) a geração de processadores de linguagens:

`antlr4`	compilação de gramáticas ANTLR-v4
`antlr4-test`	depuração de gramáticas
`antlr4-clean`	eliminação dos ficheiros gerados pelo ANTLR-v4
`antlr4-main`	geração da classe `main` para a gramática
`antlr4-visitor`	geração de uma classe *visitor* para a gramática
`antlr4-listener`	geração de uma classe *listener* para a gramática
`antlr4-build`	compila gramáticas e o código `java` gerado
`antlr4-run`	executa a classe `*Main` associada à gramática
`antlr4-jar-run`	executa um ficheiro jar (incluíndo os jars do antlr)
`antlr4-javac`	compilador java (jar do antlr no CLASSPATH)
`antlr4-java`	máquina virtual java (jar do antlr no CLASSPATH)
`java-clean`	eliminação dos ficheiros binários `java`
`view-javadoc`	abre a documentação de uma classe `java` no *browser*
`st-groupfile2string`	converte um `STGroupFile` num `STGroupString`

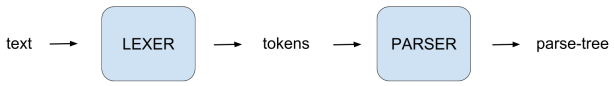
- Estes comandos estão disponíveis no *elearning* e fazem parte da instalação automática.

2 Exemplos

2.1 *Hello*

ANTLR4: *Hello*

- `ANTLR4`:



- Exemplo:

```
// (this is a line comment)
grammar Hello;                    // Define a grammar called Hello
// parser (first letter in lower case):
r : 'hello' ID ;                  // match keyword hello followed by an identifier
// lexer (first letter in upper case):
ID : [a-z]+ ;                     // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ;         // skip spaces, tabs, newlines, (Windows)
```

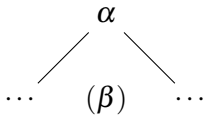
- As duas gramáticas – lexical e sintáctica – são expressas com instruções com a seguinte estrutura:

$$\alpha : \beta;$$

em que α corresponde a um único símbolo lexical ou sintáctico (dependendo da sua primeira letra ser, respectivamente, maiúscula ou minúscula); e em que β é uma expressão simbólica equivalente a α.

ANTLR4: *Hello* (2)

- Uma sequência de símbolos na entrada que seja reconhecido por esta regra gramatical pode sempre ser expressa por uma estrutura tipo árvore (chamada *sintáctica*), em que a raiz corresponde a α e os ramos à sequência de símbolos expressos em β:



- Podemos agora gerar o processador desta linguagem e experimentar a gramática utilizando o programa de teste do `ANTLR4`.

```
antlr4 Hello.g4

antlr4-javac Hello*.java

echo "hello compiladores" | antlr4-test Hello r -tokens
```

- Utilização:

```
antlr4-test [<Grammar> <rule>] [-tokens | -tree | -gui]
```

ANTLR4: Ficheiros gerados

- Executando o comando `antlr4` sobre esta gramática obtemos os seguintes ficheiros:

- Ficheiros gerados:
  - `HelloLexer.java`: código `Java` com a análise léxica (gera *tokens* para a análise sintáctica)
  - `Hello.tokens` e `HelloLexer.tokens`: ficheiros com a identificação de *tokens* (pouco importante nesta fase, mas serve para modularizar diferentes analisadores léxicos e/ou separar a análise léxica da análise sintáctica)
  - `HelloParser.java`: código `Java` com a análise sintáctica (gera a árvore sintáctica do programa)
  - `HelloListener.java` e `HelloBaseListener.java`: código `Java` que implementa automaticamente um padrão de execução de código tipo *listener* (*observer*, *callbacks*) em todos os pontos de entrada e saída de todas as regras sintácticas do compilador.
- Podemos executar o `ANTLR4` com a opção `-visitor` para gerar também código `Java` para o padrão tipo *visitor* (difere do *listener* porque a visita tem de ser explicitamente requerida).
  - `HelloVisitor.java` e `HelloBaseVisitor.java`: código `Java` que implementa automaticamente um padrão de execução de código tipo *visitor* todos os pontos de entrada e saída de todas as regras sintácticas do compilador.

2.2 *Expr*

ANTLR4: `Expr`

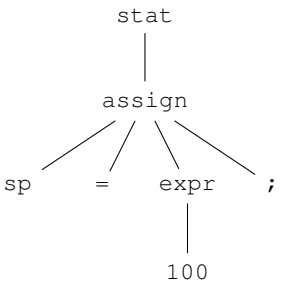
- Exemplo:

```
grammar Expr;
stat: assign ;
assign: ID '=' expr ';' ;
expr: INT ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

- Se executarmos o compilador criado com a entrada:

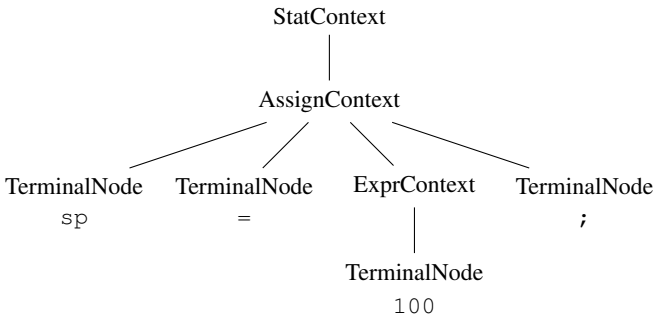
```
sp = 100;
```

- Vamos obter a seguinte árvore sintáctica:



ANTLR4: contexto automático

- Para facilitar a análise semântica e a síntese, o `ANTLR4` tenta ajudar na resolução automática de muitos problemas (como é o caso dos *visitors* e dos *listeners*)
- No mesmo sentido são geradas classes (e em execução os respectivos objectos) com o contexto de todas as regras da gramática:



ANTLR4: contexto automático (2)

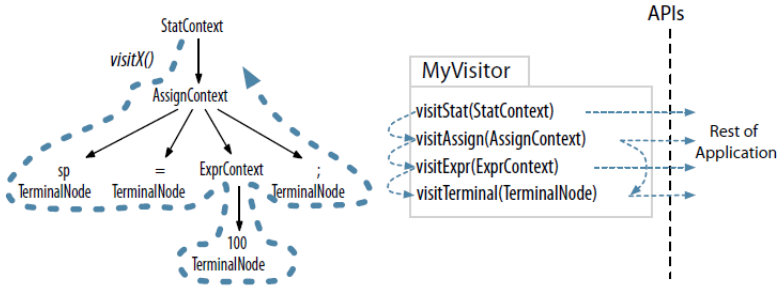
```
grammar Expr;                          → classes: ExprLexer and ExprParser
stat : assign ;                        → class StatContext in ExprParser
assign : ID '=' expr ';' ;             → class AssignContext in ExprParser
expr : INT ;                           → class ExprContext in ExprParser

ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

```
public class ExprParser extends Parser {
  public static class StatContext extends ParserRuleContext {
    public AssignContext assign() {
      ...
    }
    ...
  }
  ...
}
```

ANTLR4: *visitor*

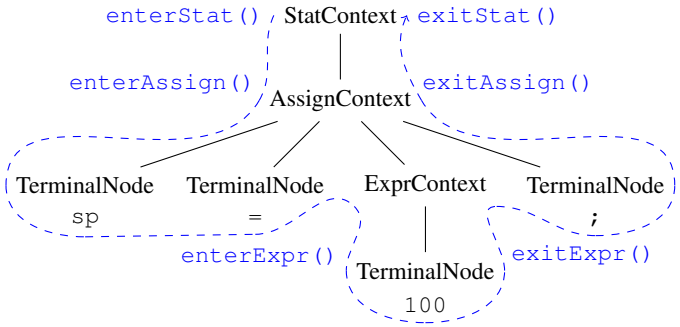
- Os objectos de contexto têm a si associada toda a informação relevante da análise sintáctica (*tokens*, referência aos nós filhos da árvore, etc.)
- Por exemplo o contexto `AssignContext` contém métodos `ID` e `expr` para aceder aos respectivos nós.

- No caso do código gerado automaticamente do tipo *visitor* o padrão de invocação é ilustrado a seguir:



ANTLR4: *listener*

- O código gerado automaticamente do tipo *listener* tem o seguinte padrão de invocação:



- A sua ligação à restante aplicação é a seguinte:

ANTLR4: atributos e acções

- É possível associar *atributos* e *acções* às regras:

```
grammar ExprAttr;
stat: assign ;
assign: ID '=' e=expr ';'
    {System.out.println($ID.text+" = "+$e.v);} // action
    ;
expr returns[int v]: INT  // result attribute named v in expr
    {$v = Integer.parseInt($INT.text);}       // action
    ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

- Ao contrário dos *visitors* e *listeners*, a execução das acções ocorre durante a análise sintáctica.
- A execução de cada acção ocorre no contexto onde ela é declarada. Assim se uma acção estiver no fim de uma regra (como exemplificado acima), a sua execução ocorrerá após o respectivo reconhecimento.

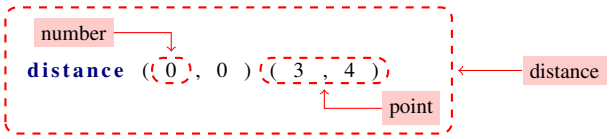
- A linguagem a ser executada na acção não tem de ser necessariamente Java (existem muitas outras possíveis, como C++ e python).
- Também podemos passar atributos para a regra (tipo passagem de argumentos para um método):

```
assign: ID '=' e=expr[true] ';' // argument passing to expr
   {System.out.println($ID.text+" = "+$e.v);}
   ;
expr[boolean a]      // argument attribute named a in expr
   returns[int v]:   // result attribute named v in expr
   INT {
      if ($a)
         System.out.println("Wow! Used in an assignment!");
      $v = Integer.parseInt($INT.text);
   } ;
```

- É clara a semelhança com a passagem de argumentos e resultados de métodos.
- Diz que os atributos são *sintetizados* quando a informação provém de sub-regras, e *herdados* quando se envia informação para sub-regras.

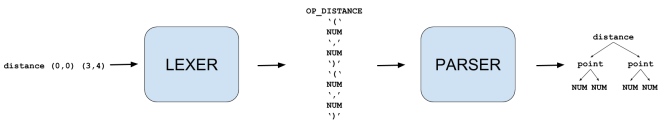
2.3 Exemplo figuras

- Recuperando o exemplo das figuras.



- Gramática inicial para figuras:

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



Integração num programa

```
import java.io.IOException;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
public class ShapesMain {
   public static void main(String[] args) {
      try {
         // create a CharStream that reads from standard input:
         CharStream input = CharStreams.fromStream(System.in);
         // create a lexer that feeds off of input CharStream:
         ShapesLexer lexer = new ShapesLexer(input);
         // create a buffer of tokens pulled from the lexer:
         CommonTokenStream tokens = new CommonTokenStream(lexer);
         // create a parser that feeds off the tokens buffer:
         ShapesParser parser = new ShapesParser(tokens);
         // begin parsing at distance rule:
         ParseTree tree = parser.distance();
         if (parser.getNumberOfSyntaxErrors() == 0) {
            // print LISP-style tree:
            // System.out.println(tree.toStringTree(parser));
         }
      }
      catch(IOException e) {
         e.printStackTrace();
         System.exit(1);
```

```
        }
        catch(RecognitionException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

- O comando `antlr4-main` gera automaticamente esta classe com uma primeira implementação do método `main`.

2.4 Exemplo *visitor*

- Uma primeira versão (limpa) de um *visitor* pode ser gerada com o script `antlr4-visitor`
- Depois podemos alterá-la, por exemplo, da seguinte forma:

```
import org.antlr.v4.runtime.tree.AbstractParseTreeVisitor;

public class ShapesMyVisitor extends ShapesBaseVisitor<Object> {
  @Override
  public Object visitDistance(ShapesParser.DistanceContext ctx) {
    double res;
    double[] p1 = (double[]) visit(ctx.point(0));
    double[] p2 = (double[]) visit(ctx.point(1));
    res = Math.sqrt(Math.pow(p1[0]-p2[0],2) +
                    Math.pow(p1[1]-p2[1],2));
    System.out.println("visitDistance: "+res);
    return res;
  }

  @Override
  public Object visitPoint(ShapesParser.PointContext ctx) {
    double[] res = new double[2];
    res[0] = Double.parseDouble(ctx.x.getText());
    res[1] = Double.parseDouble(ctx.y.getText());

    return (Object)res;
  }
}
```

- Para utilizar esta classe:

```
public static void main(String[] args) {
    ...
      // visitor:
    ShapesMyVisitor visitor = new ShapesMyVisitor();
    System.out.println("distance: "+visitor.visit(tree));
    ...
}
```

- O comando `antlr4-main` permite a geração automática deste código no método `main`.

```
antlr4-main <Grammar> <start-rule> -v <nome-da-classe-ou-ficheiro-visitor> ...
```

- Note que podemos criar o método `main` com os *listeners* e *visitors* que quisermos (a ordem especificada nos argumentos do comando é mantida).

2.5 Exemplo *listener*

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

import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.tree.ErrorNode;
import org.antlr.v4.runtime.tree.TerminalNode;

public class ShapesMyListener extends ShapesBaseListener {
    @Override
    public void enterPoint(ShapesParser.PointContext ctx) {
        int x = Integer.parseInt(ctx.x.getText());
        int y = Integer.parseInt(ctx.y.getText());
```

```
        out.println("enterPoint x="+x+",y="+y);
    }

    @Override
    public void exitPoint(ShapesParser.PointContext ctx) {
        int x = Integer.parseInt(ctx.x.getText());
        int y = Integer.parseInt(ctx.y.getText());
        out.println("exitPoint x="+x+",y="+y);
    }
}
```

- Para utilizar esta classe:

```
public static void main(String[] args) {
    ...
      // listener:
    ParseTreeWalker walker = new ParseTreeWalker();
    ShapesMyListener listener = new ShapesMyListener();
    walker.walk(listener, tree);
    ...
}
```

- O comando `antlr4-main` permite a geração automática deste código no método `main`.

```
antlr4-main <Grammar> <start-rule> -l <nome-da-classe-ou-ficheiro-listener> ...
```

3 Construção de gramáticas

- A construção de gramáticas pode ser considerada uma forma de *programação simbólica*, em que existem símbolos que são equivalentes a sequências (que façam sentido) de outros símbolos (ou mesmo dos próprios).
- Os símbolos utilizados dividem-se em *símbolos terminais e não terminais*.
- Os símbolos terminais (ou *tokens*) são predefinidos, ou definidos fora da gramática; e os símbolos não terminais são definidos por produções (regras) da gramática (sendo estas transformações equivalentes de uma sequência de símbolos noutra sequência).
- No fim, todos os símbolos não terminais, com mais ou menos transformações, devem poder ser expressos em símbolos terminais.
- Uma gramática é construída especificando as *regras* ou produções dos elementos gramaticais.

```
grammar SetLang;      // a grammar example
stat: set set;        // stat is a sequence of two 'set'
set: '{' elem* '}';   // set is zero or more 'elem' inside '{' '}'
elem: ID | NUM;
ID: [a-z]+;
NUM: [0-9]+;
```

- Sendo a sua construção uma forma de programação, podemos beneficiar da identificação e reutilização de padrões comuns de resolução de problemas.

- Surpreendentemente, o número de padrões base é relativamente baixo:

  1. *Sequência*: sequência de elementos;
  2. *Optativo*: aplicação optativa do elemento (zero ou uma ocorrência);
  3. *Repetitivo*: aplicação repetida do elemento (zero ou mais, uma ou mais);
  4. *Alternativa*: escolha entre diferentes alternativas (como por exemplo, diferentes tipos de instruções);
  5. *Recursão*: definição directa ou indirectamente recursiva de um elemento (por exemplo, instrução condicional é uma instrução que selecciona para execução outras instruções);

- É de notar que a recursão e a iteração são alternativas entre si. Admitindo a existência da sequência vazia, os padrões optativo e repetitivo são implementáveis com recursão.
- No entanto, como em programação em geral, por vezes é mais adequado expressar recursão, e outras iteração.

- Considere o seguinte programa em Java:

```
import static java.lang.System.*;
public class PrimeList {
   public static void main(String[] args) {
      if (args.length != 1) {
         out.println("Usage: PrimeList -ea <n>");
         exit(1);
      }
      int n = 0;
      try {
         n = Integer.parseInt(args[0]);
      }
      catch(NumberFormatException e) {
         out.println("ERROR: invalid argument \""+args[0]+"\"");
         exit(1);
      }
      for(int i = 2;i <= n;i++)
         if(isPrime(i))
            out.println(i);
   }

   public static boolean isPrime(int n) {
      assert n > 1; // precondition

      boolean result = (n == 2 || n % 2 != 0);
      for(int i = 3;result && (i*i <= n);i+=2)
         result = (n % i != 0);
      return result;
   }
}
```

- Mesmo na ausência de uma gramática definida explicitamente, podemos neste programa inferir todos os padrões atrás referidos:
  1. *Sequência*: a instrução atribuição de valor é definida como sendo um identificador, seguido do carácter =, seguido de uma expressão.
  2. *Optativo*: a instrução condicional pode ter, ou não, a selecção de código para a condição falsa.
  3. *Repetitivo*: (1) uma classe é uma repetição de membros; (2) um algoritmo é uma repetição de comandos.
  4. *Alternativa*: diferentes instruções podem ser utilizadas onde uma instrução é esperada.
  5. *Recursão*: a instrução composta é definida como sendo uma sequência de instruções delimitada por chavetas; qualquer uma dessas instruções pode ser também uma instrução composta.

3.1 Especificação de gramáticas

- Uma linguagem para especificação de gramáticas precisa de suportar este conjunto de padrões.
- Para especificar elementos léxicos (*tokens*) a notação utilizada assenta em *expressões regulares*.
- A notação tradicionalmente utilizada para a análise sintáctica denomina-se por `BNF` (*Backus-Naur Form*).

```
<symbol> ::= <meaning>
```

- Esta última notação teve origem na construção da linguagem `Algol` (1960).
- O `ANTLR4` utiliza uma variação alterada e aumentada (`Extended BNF` ou `EBNF`) desta notação onde se pode definir construções opcionais e repetitivas.

```
<symbol> : <meaning> ;
```

4 ANTLR4: Estrutura léxica

4.1 Comentários

- A estrutura léxica do `ANTLR4` deverá ser familiar para a maioria dos programadores já que se aproxima da sintaxe das linguagens da família do `C` (`C++`, `Java`, etc.).

- Os comentários são em tudo semelhantes aos do Java permitindo a definição de comentários de linha, multilinha, ou tipo `JavaDoc`.

```
/**
 * Javadoc alike comment!
 */
grammar Name;
/*
multiline comment
*/

/** parser rule for an identifier */
id: ID ; // match a variable name
```

4.2 Identificadores

- O primeiro caráter dos identificadores tem de ser uma letra, seguida por outras letras dígitos ou o caráter _
- Se a primeira letra do identificador é minúscula, então este identificador representa uma regra sintáctica; caso contrário (i.e. letra maiúscula) então estamos na presença duma regra léxica.

```
ID, LPAREN, RIGHT_CURLY, Other // lexer token names
expr, conditionalStatment      // parser rule names
```

- Como em `Java`, podem ser utilizados caracteres `Unicode`.

4.3 Literais

- Em `ANTLR4` não há distinção entre literais do tipo carácter e do tipo *string*.
- Todos os literais são delimitados por aspas simples.
- Exemplos: `'if'`, `'>='`, `'assert'`
- Como em `Java`, os literais podem conter sequências de escape tipo `Unicode` (`'\u3001'`), assim como as sequências de escape habituais (`'\'\r\t\n'`)

4.4 Palavras reservadas

- O `ANTLR4` tem a seguinte lista de palavras reservadas (i.e. que não podem ser utilizadas como identificadores):

```
import, fragment, lexer,
parser, grammar, returns,
locals, throws, catch,
finally, mode, options,
tokens, skip
```

- Mesmo não sendo uma palavra reservada, não se pode utilizar a palavra `rule` já que esse nome entra em conflito com os nomes gerados no código.

4.5 Acções

- As acções são blocos de código escritos na linguagem destino (`Java` por omissão).
- As acções podem ter múltiplas localizações dentro da gramática, mas a sintaxe é sempre a mesma: texto arbitrário delimitado por chavetas: `{...}`
- Se por caso existirem *strings* ou comentários (ambos tipo `C/Java`) contendo chavetas não há necessidade de incluir um caráter de escape (`{..."}"./*}*/..}`).
- O mesmo acontece se as chavetas foram balanceadas (`{{...{}...}}`).
- Caso contrário, tem de se utilizar o caráter de escape (`{\{}, {\}}`).
- O texto incluído dentro das acções tem de estar conforme com a linguagem destino.

- As acções podem aparecer nas regras léxicas, nas regras sintácticas, na especificação de excepções da gramática, nas secções de atributos (resultado, argumento e variáveis locais), em certas secções do cabeçalho da gramática e em algumas opções de regras (predicados semânticos).

- Pode considerar-se que cada acção será executada no contexto onde aparece (por exemplo, no fim do reconhecimento duma regra).

```
grammar Expr;
stat:
    {System.out.println("[stat]: before assign");} assign
  | expr {System.out.println("[stat]: after expr");}
  ;
assign:
  ID
  {System.out.println("[assign]: after ID and before =!");}
  '=' expr ';';
expr: INT {System.out.println("[expr]: INT!");} ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

5 ANTLR4: Regras léxicas

- A gramática léxica é composta por regras (ou produções), em que cada regra define um *token*.
- As regras léxicas têm de começar por uma letra maiúscula, e podem ser visíveis apenas dentro do analisador léxico:

```
INT: DIGIT+   ;          // visible in both parser and lexer
fragment DIGIT: [0-9]; // visible only in lexer
```

- Como, por vezes, a mesma sequência de caracteres pode ser reconhecida por diferentes regras (por exemplo: identificadores e palavras reservadas), o ANTLR4 estabelece critérios que permitem eliminar esta ambiguidade (e dessa forma, reconhecer um, e um só, *token*).

- Esses critérios são essencialmente dois (na ordem seguinte):

  1. Reconhece *tokens* que consomem o máximo possível de caracteres.

     Por exemplo, num reconhecedor léxico para Java, o texto `ifa` é reconhecido com um único *token* tipo identificador, e não como dois *tokens* (palavra reservada `if` seguida do identificador `a`).

  2. Dá prioridade às regras definidas em primeiro lugar.

     Por exemplo, na gramática seguinte:

     ```
     ID: [a-z]+;
     IF: 'if';
     ```

     o *token* `IF` nunca vai ser reconhecido!

- O ANTLR4 também considera que os *tokens* definidos implicitamente em regras sintácticas, estão definidos *antes* dos definidos explicitamente por regras léxicas.
- A especificação destas regras utiliza *expressões regulares*.

Expressões regulares em ANTLR4

Syntax	*Description*
$R : \ldots ;$	*Define lexer rule R*
X	*Match lexer rule element X*
$'literal'$	*Match literal text*
$[char\text{-}set]$	*Match one of the chars in char-set*
$'x'..'y'$	*Match one of the chars in the interval*
$X\,Y\,\ldots\,Z$	*Match a sequence of rule lexer elements*
$(\ldots)$	*Lexer subrule*
$X?$	*Match rule element X*
$X*$	*Match rule element X zero or more times*
$X+$	*Match rule element X one or more times*
$\sim x$	*Match one of the chars NOT in the set defined by x*
.	*Match any char*
$X*?Y$	*Match X until Y appears (non-greedy match)*
$\{\ldots\}$	*Lexer action*
$\{p\}?$	*Evaluate semantic predicate p (if false, the rule is ignored)*
$x\vert\ldots\vert z$	*Multiple alternatives*

5.1 Padrões léxicos típicos

Token category	*Possible implementation*

Identifiers	`ID: LETTER (LETTER | DIGIT)*;`
	`fragment LETTER: 'a'..'z'|'A'..'Z'|'_';`
	`  // same as: [a-zA-Z_]`
	`fragment DIGIT: '0'..'9';`
	`  // same as: [0-9]`

Numbers	`INT: DIGIT+;` `FLOAT: DIGIT+ '.' DIGIT+ | '.' DIGIT+;`
Strings	`STRING: '"' (ESC | . )*? '"';` `fragment ESC: '\\"' | '\\\\' ;`
Comments	`LINE_COMMENT: '//' .*? '\n' -> skip;` `COMMENT: '/*' .*? '*/' -> skip;`
Whitespace	`WS: [ \t\n\r]+ -> skip;`

5.2 Operador léxico "não ganancioso"

- Por omissão, a análise léxica é "gananciosa".
- Isto é, os *tokens* são gerados com o maior tamanho possível.
- Esta particularidade é em geral a desejada, mas pode trazer problemas em alguns casos.
- Por exemplo, se quisermos reconhecer um *string*:

```
STRING: '"' .* '"';
```

- (No analisador léxico o ponto (`.`) reconhece qualquer carácter excepto o `EOF`.)
- Esta regra não funciona, porque, uma vez reconhecido o primeiro carácter `"`, o analisador léxico vai reconhecer todos os caracteres como pertencendo ao `STRING` até ao último carácter `"`.
- Este problema resolve-se com o operador *non-greedy*:

```
STRING: '"' .*? '"'; // match all chars until a " appears!
```

6 ANTLR4: Estrutura sintáctica

- As gramáticas em `ANTLR4` têm a seguinte estrutura sintáctica:

```
grammar Name;          // mandatory
options { ... }        // optional
import ... ;           // optional
tokens { ... }         // optional
@actionName { ... }    // optional
rule1 : ... ;          // parser and lexer rules
...
```

- As regras léxicas e sintácticas pode aparecer misturadas e distinguem-se por a primeira letra do nome da regra ser minúscula (analisador sintáctico), ou maiúscula (analisador léxico).
- Como já foi referido, a ordem pela qual as regras léxicas são definidas é muito importante.
- É possível separar as gramáticas sintácticas das léxicas precedendo a palavra reservada `grammar` com as palavras reservadas `parser` ou `lexer`.

```
parser grammar NameParser;
...

lexer grammar NameLexer;
...
```

- A secção das *opções* permite definir algumas opções para os analisadores (e.g. origem dos *tokens*, e a linguagem de programação de destino).

```
options { tokenVocab=NameLexer; }
```

- Qualquer opção pode ser redefinida por argumentos na invocação do `ANTLR4`.
- A secção de **`import`** relaciona-se com herança de gramáticas (que veremos mais à frente).

6.1 Secção de *tokens*

- A secção de *tokens* permite associar identificadores a *tokens*.
- Esses identificadores devem depois ser associados a regras léxicas, que podem estar na mesma gramática, noutra gramática, ou mesmo ser directamente programados.

```
tokens { «Token1», ..., «TokenN» }
```

- Por exemplo: **`tokens`** `{ BEGIN, END, IF, ELSE, WHILE, DO }`
- Note que não é necessário ter esta secção quando os tokens tem origem numa gramática lexical antlr4 (basta a secção `options` com a variável `tokenVocab` correctamente definida).

6.2 Acções no preâmbulo da gramática

- Esta secção permite a definição de *acções* no preâmbulo da gramatica (como já vimos, também podem existem acções noutras zonas da gramática).
- Actualmente só existem dois acções possíveis nesta zona (com o `Java` como linguagem destino): `header` e `members`

```
grammar Count;
@header {
package foo;
}
@members {
int count = 0;
}
```

- A primeira injecta código no inicio de ficheiros, e a segunda permite que se acrescente membros às classes do analisador sintáctico e/ou léxico.
- Eventualmente podemos restringir estas acções ou ao analisador sintáctico (`@parser::header`) ou ao analisador léxico (`@lexer::members`)

7 ANTLR4: Regras sintácticas

Construção de regras: síntese

Syntax	*Description*
r : ... ;	*Define rule r*
x	*Match rule element x*
x y ... z	*Match a sequence of rule elements*
(...)	*Subrule*
x?	*Match rule element x*
x∗	*Match rule element x zero or more times*
x+	*Match rule element x one or more times*
x | ... | *z*	*Multiple alternatives*

A rule element is a token (lexical, or terminal rule), a syntactical rule (non-terminal), or a subrule.

Regras sintácticas: movendo informação

- Em ANTLR4 cada regra sintáctica pode ser vista como uma espécie de método, podendo-se havendo mecanismos de comunicação similares: *argumentos* e *resultado*, assim como *variáveis locais* à regra.
- Podemos também anotar regras com um nome alternativo:

```
expr: e1=expr '+' e2=expr
    | INT;
```

- Podemos também etiquetar com nomes, diferentes alternativas duma regra:

```
expr: expr '*' e2=expr # ExprMult
    | expr '+' e2=expr # ExprAdd
    | INT              # ExprInt
    ;
```

- O ANTLR4 irá gerar informação de contexto para cada nome (incluindo métodos para usar no *listener* e/ou nos *visitors*).

```
grammar Info;

@header {
import static java.lang.System.*;
}

main: seq1=seq[true] seq2=seq[false] {
      out.println("average(seq1): "+$seq1.average);
      out.println("average(seq2): "+$seq2.average);
   }
   ;

seq[boolean crash] returns[double average=0]
   locals[int sum=0, int count=0]:
   '(' ( INT {$sum+=$INT.int;$count++;} )* ')' {
      if ($count > 0)
         $average = (double)$sum/$count;
      else if ($crash) {
         err.println("ERROR: divide by zero!");
         exit(1);
      }
   }
  ;

INT: [0-9]+;
WS: [ \t\n\r]+ -> skip;
```

7.1 Padrões sintácticos típicos

Pattern name	*Possible implementation*

Sequence	`x y ... z`
	`'[' INT+ ']'`
	`'[' INT* ']'`

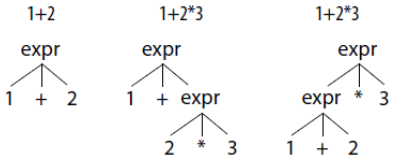
Sequence with terminator	`( instruction ';' )* // program sequence` `( row '\n' )*  // lines of data`
Sequence with separator	`expr ( ',' expr)*  // function call arguments` `( expr ( ',' expr)* )? // optional arguments`
Choice	`type: 'int' | 'float';` `instruction: conditional | loop | ... ;`

Token dependence	`'(' expr ')'  // nested expression`
	`ID '[' expr ']'  // array index`
	`'{' instruction+ '}' // compound instruction`
	`'<' ID (',' ID)* '>' // generic type specifier`

Recursivity	`expr: '(' expr ')' | ID;`
	`classDef: 'class' ID`
	`  '{' (classDef|method|field)* '}';`

7.2 Precedência

- Por vezes, formalmente, a interpretação da ordem de aplicação de operadores pode ser subjectiva:



- Em ANTLR4 esta ambiguidade é resolvida dando primazia às sub-regras declaradas primeiro:

```
expr: expr '*' expr // higher priority
    | expr '+' expr
    | INT           // lower priority
    ;
```

7.3 Associatividade

- Por omissão, a associatividade na aplicação do (mesmo) operador é feita da esquerda para a direita:
  $a+b+c = ((a+b)+c)$
- No entanto, há operadores, como é o caso da potência, que podem requerer a associatividade inversa:
  $a \uparrow b \uparrow c = a^{b^c} = a^{(b^c)}$
- Este problema é resolvido em ANTLR4 de seguinte forma:

```
expr: <assoc=right> expr '^' expr
    | expr '*' expr // higher priority
    | expr '+' expr
    | INT           // lower priority
    ;
```

7.4 Herança de gramáticas

- A secção de *import* implementa um mecanismo de herança entre gramáticas.

- Por exemplo as gramáticas:

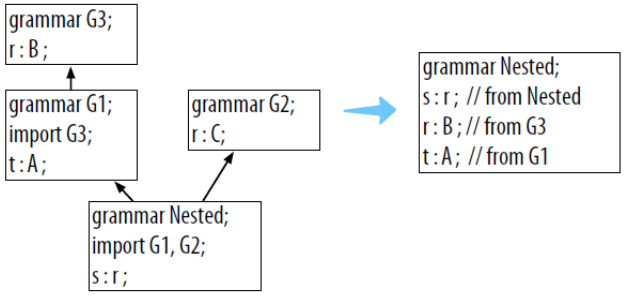
```
grammar ELang;
stat : (expr ';')* EOF ;
expr : INT ;
INT : [0-9]+ ;
WS : [ \r\t\n]+ -> skip ;
```

```
grammar MyELang;
import ELang;
expr : INT | ID ;
ID : [a-z]+ ;
```

- Geram a gramática `MyELang` equivalente:

```
grammar MyELang;
stat : (expr ';')+ EOF ;
expr : INT | ID ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \r\t\n]+ -> skip ;
```

- Isto é, as regras são herdadas, excepto quando são redefinidas na gramática descendente.
- Este mecanismo permite herança múltipla:



- Note-se a importância na ordem dos `imports` na gramática Nested.
- A regra *r* vem da gramática G3 e não da gramática G2.

8 ANTLR4: outras funcionalidades

8.1 Mais sobre acções

- Já vimos que é possível acrescentar directamente na gramática acções (expressas na linguagem destino) que são executadas durante a fase de análise sintáctica (na ordem expressa na gramática).
- Podemos também associar a cada regra dois blocos especiais de código – `@init` e `@after` – cuja execução, respectivamente, precede ou sucede ao reconhecimento da regra.
- O bloco `@init` pode ser útil, por exemplo, para inicializar variáveis.
- O bloco `@after` é uma alternativa a colocar a acção no fim da regra.

8.2 Exemplo: tabelas CSV

Exemplo

- Exemplo: gramática para ficheiros tipo CSV com os seguintes requisitos:
  1. A primeira linha indica o nome dos campos (deve ser escrita sem nenhuma formatação em especial);
  2. Em todas as linhas que não a primeira associar o valor ao nome do campo (devem ser escritas com a associação explicita, tipo atribuição de valor com `field = value`.

```
grammar CSV;

file: line line* EOF;

line: field (SEP field)* '\r'? '\n';

field: TEXT | STRING | ;
```

```
SEP: ','; // (' ' | '\t')*
STRING: [ \t]* '"' .*? '"' [ \t]*;
TEXT: ~[,"\r\n]~[,\r\n]*;
```

Exemplo

```
grammar CSV;
@header {
import static java.lang.System.*;
}
@parser::members {
   protected String[] names = new String[0];
   public int dimNames() { ··· }
   public void addName(String name) { ··· }
   public String getName(int idx) { ··· }
}

file: line[true] line[false]* EOF;

line[boolean firstLine]
   locals[int col = 0]
   @after { if (!firstLine) out.println(); }
   : field[$firstLine,$col++] (SEP field[$firstLine,$col++])* '\r'? '\n';

field[boolean firstLine, int col]
   returns[String res = ""]
   @after {
      if ($firstLine)
         addName($res);
      else if ($col >= 0 && $col < dimNames())
         out.print("  "+getName($col)+": "+$res);
      else
         err.println("\nERROR: invalid field \""+$res+"\" in column "+($col+1));
   }
   :
   (TEXT {$res = $TEXT.text.trim();}) |
   (STRING {$res = $STRING.text.trim();}) |
   ;

SEP: ','; // (' ' | '\t')*
STRING: [ \t]* '"' .*? '"' [ \t]*;
TEXT: ~[,"\r\n]~[,\r\n]*;
```

8.3 Gramáticas ambíguas

- A definição de gramáticas presta-se, com alguma facilidade, a gerar ambiguidades.
- Esta característica nas linguagens humanas é por vezes procurada (onde estaria a literatura e a poesia se não fosse assim), mas geralmente é um problema.

  > "Para o meu orientador, para quem nenhum agradecimento é demasiado."
  >
  > "O professor falou aos alunos de engenharia"
  >
  > *"What rimes with orange? ... No it doesn't!"*

- No caso das linguagens de programação, em que os efeitos são para ser interpretados e executados por máquinas (e não por nós), não há espaço para ambiguidades.
- Assim, seja por construção da gramática, seja por regras de prioridade que lhe sejam aplicadas por omissão, as gramáticas não podem ser ambíguas.
- Em ANTLR4 a definição e construção de regras define prioridades.

Gramáticas ambíguas: analisador léxico

- Se as gramáticas léxicas fossem apenas definidas por expressões regulares que competem entre si para consumir os caracteres de entrada, então elas seriam naturalmente ambíguas.

```
...
conditional: 'if' '(' expr ')' 'then' stat; // incomplete
ID: [a-zA-Z]+;
...
```

- Neste caso a sequência de caracteres **`if`** tanto pode dar um identificador como uma palavra reservada.
- O ANTLR4 utiliza duas regras fora das expressões regulares para lidar com ambiguidade:
  1. Por omissão, escolhe o *token* que consume o máximo número de caracteres da entrada;
  2. Dá prioridade aos *tokens* definidos primeiro (sendo que os definidos implicitamente na gramática sintáctica têm precedência sobre todos os outros).

Gramáticas ambíguas: analisador sintáctico

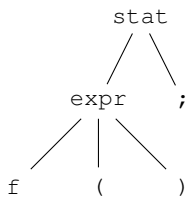
- Já vimos que nas regras sintácticas também pode haver ambiguidade.
- Os dois excertos seguintes exemplificam gramáticas ambíguas:

```
stat: ID '=' expr
    | ID '=' expr
    ;
expr: NUM
    ;
```

```
stat: expr ';'
    | ID '(' ')' ';'
    ;
expr: ID '(' ')'
    | NUM
    ;
```

- Em ambos os casos a ambiguidade resulta de ser ter uma sub-regra repetida, directamente, no primeiro caso, e indirectamente, no segundo caso.
- A gramática diz-se ambígua porque, para a mesma entrada, poderíamos ter duas árvores sintácticas diferentes.

Expressão `f();`



Instrução `f();`

- Outros exemplos de ambiguidade são os da precedência e associatividade de operadores (secções 7.2 e 7.3).
- O ANTLR4 tem regras adicionais para eliminar ambiguidades sintácticas.
- Tal como no analisador léxico, regras *Ad hoc* fora da notação das gramática independentes de contexto, garantem a não ambiguidade.
- Essas regras são as seguintes:
  1. As alternativas, directa ou indirectamente, definidas primeiro têm precedência sobre as restantes.
  2. Por omissão, a associatividade de operadores é à esquerda.
- Das duas árvores sintácticas apresentadas no exemplo anterior, a gramática definida impõe a primeira alternativa.
- A linguagem `C` tem ainda outro exemplo prático de ambiguidade.
- A expressão `i*j` tanto pode ser uma multiplicação de duas variáveis, como a declaração de uma variável `j` como ponteiro para o tipo de dados `i`.
- Estes dois significados tão diferentes podem também ser resolvidos em gramáticas ANTLR4 com os chamados *predicados semânticos*.

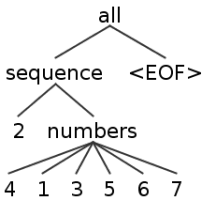
8.4 Predicados semânticos

- Em `ANTLR4` é possível utilizar informação semântica (expressa na linguagem destino e injetada na gramática), para orientar o analisador sintáctico.
- Essa funcionalidade chama-se *predicados semânticos*: `{...}?`
- Os predicados semânticos permitem seletivamente activar/desactivar porções das regras gramaticais durante a própria análise sintáctica.
- Vamos, como exemplo, desenvolver uma gramática para analisar sequências de números inteiros, mas em que o primeiro número não pertence à sequência, mas indica sim a dimensão da sequência:
- Assim a lista `2 4 1 3 5 6 7` indicaria duas sequências: `(4,1) (5,6,7)`

Exemplo

```
grammar Seq;

all: sequence* EOF;

sequence: INT numbers;

numbers: INT*;

INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

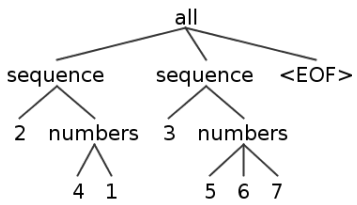
Com esta gramática, a árvore sintáctica gerada para a entrada `2 4 1 3 5 6 7` é:



Exemplo

```
grammar Seq;

all: sequence* EOF;

sequence
    @init { System.out.print("("); }
    @after { System.out.println(")"); }
    : INT numbers[$INT.int];

numbers[int count] locals [int c = 0]
    : ( {$c < $count}? INT
        {$c++; System.out.print(($c == 1 ? "" : " ")+$INT.text);}
      )* ;

INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

Agora a árvore sintáctica já corresponde ao pretendido:



8.5 Separar analisador léxico do analisador sintáctico

- Muito embora se possa definir a gramática completa, juntando a análise léxica e a sintáctica no mesmo módulo, podemos também separar cada uma dessas gramáticas.
- Isso facilita, por exemplo, a reutilização de analisadores léxicos.
- Existem também algumas funcionalidades do analisador léxico, que obrigam a essa separação ("ilhas" lexicais).
- Para que a separação seja bem sucedida há um conjunto de regras que devem ser seguidas:

  1. Cada gramática indica o seu tipo no cabeçalho:
  2. Os nomes das gramáticas devem (respectivamente) terminar em `Lexer` e `Parser`
  3. Todos os *tokens* implicitamente definidos no analisador sintáctico têm de passar para o analisador léxico (associando-lhes um identificador para uso no *parser*).
  4. A gramática do analisador léxico deve ser compilada pelo `ANTLR4` antes da gramática sintáctica.
  5. A gramática sintáctica tem de incluir uma opção (`tokenVocab`) a indicar o analisador léxico.

```
lexer grammar NAMELexer;
...

parser grammar NAMEParser;
options {
   tokenVocab=NAMELexer;
}
...
```

- No teste da gramática deve utilizar-se o nome sem o sufixo:

```
antlr4-test NAME rule
```

Exemplo

```
lexer grammar CSVLexer;

COMMA: ',';
EOL: '\r'? '\n';
STRING: '"' ( '""' | ~'"' )* '"';
TEXT: ~[,"\r\n]~[,\r\n]*;

parser grammar CSVParser;

options {
   tokenVocab=CSVLexer;
}

file: firstRow row* EOF;

firstRow: row;

row: field (COMMA field)* EOL;

field: TEXT | STRING | ;


antlr4 CSVLexer.g4
antlr4 CSVParser.g4
antlr4-javac CSV*java
// ou apenas: antlr4-build
antlr4-test CSV file
```

8.6 "Ilhas" lexicais

- Outra característica do ANTLR4 é a possibilidade de reconhecer um conjunto diferente de *tokens* consoante determinados critérios.
- Para esse fim existem os chamados *modos* lexicais.
- Por exemplo, em XML, o tratamento léxico do texto deve ser diferente consoante se está dentro duma "marca" (*tag*) ou fora.
- Uma restrição desta funcionalidade é o facto de só se poderem utilizar modos lexicais em gramáticas léxicas.
- Ou seja, torna-se obrigatória a separação entre os dois tipos de gramáticas.
- Os modos lexicais são geridos pelos comandos: `mode(NAME)`, `pushMode(NAME)`, `popMode`
- O modo lexical por omissão é designado por: `DEFAULT_MODE`

Exemplo

```
lexer grammar ModesLexer;

// default mode

ACTION_START: '{' -> mode(INSIDE_ACTION);
OUTSIDE_TOKEN: ~'{'+;

mode INSIDE_ACTION;
ACTION_END: '}' -> mode(DEFAULT_MODE);
INSIDE_TOKEN: ~'}'+;

parser grammar ModesParser;

options {
   tokenVocab=ModesLexer;
}

all: ( ACTION_START | OUTSIDE_TOKEN | ACTION_END |
       INSIDE_TOKEN)* EOF;
```

```
lexer grammar ModesLexer;

// default mode

ACTION_START: '{' -> pushMode(INSIDE_ACTION);
OUTSIDE_TOKEN: ~'{'+;

mode INSIDE_ACTION;
ACTION_END: '}' -> popMode;
INSIDE_ACTION_START: '{' -> pushMode(INSIDE_ACTION);
INSIDE_TOKEN: ~[{}]+;

parser grammar ModesParser;

options {
   tokenVocab=ModesLexer;
}

all: ( ACTION_START | OUTSIDE_TOKEN | ACTION_END |
       INSIDE_ACTION_START | INSIDE_TOKEN)* EOF;
```

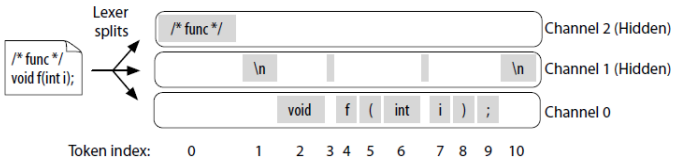
8.7 Enviar *tokens* para canais diferentes

- Nos exemplos de gramáticas que temos vindo a apresentar, tem-se optado pela acção `skip` quando na presença dos chamados espaços em branco ou de comentários.
- Esta acção faz desaparecer esses *tokens* simplificando a análise sintáctica.
- O preço a pagar (geralmente irrelevante) é perder o texto completo que lhes está associado.
- No entanto, em ANTLR4 é possível ter dois em um. Isto é, retirar *tokens* da analise sintáctica, sem no entanto fazer desaparecer completamente esses *tokens* (podendo-se recuperar o texto que lhe está associado).

- Esse é o papel dos chamados *canais léxicos*.

```
WS: [ \t\n\r]+          -> skip; // make token disappear
COMMENT: '/*' .*? '*/' -> skip; // make token disappear

WS: [ \t\n\r]+          -> channel(1); // redirect to channel 1
COMMENT: '/*' .*? '*/' -> channel(2); // redirect to channel 2
```

- A classe `CommonTokenStream` encarrega-se de juntar os tokens de todos os canais (o visível – canal zero – e os escondidos).



- (É possível ter código para aceder aos *tokens* de um canal em particular.)

Exemplo: declaração de função

```
grammar Func;

func: type=ID function=ID '(' varDecl* ')' ';';
varDecl: type=ID variable=ID;

ID: [a-zA-Z_]+;
WS: [ \t\r\n]+ -> channel(1);
COMMENT: '/*' .*? '*/' -> channel(2);
```

8.8 Reescrever a entrada

- O `ANTLR4` facilita a geração de código que resulte de uma reescrita do código de entrada. Isto é, inserir, apagar, e/ou modificar partes desse código.
- Para esse fim existe a classe `TokenStreamRewriter` (que têm métodos para inserir texto antes ou depois de *tokens*, ou para apagar ou substituir texto).
- Vamos supor que se pretende fazer algumas alterações de código fonte `Java`, por exemplo, acrescentar um comentário imediatamente antes da declaração de uma classe..
- Podemos ir buscar a gramática disponível para a versão 8 do `Java`: `Java8.g4`
  (procurar em: `https://github.com/antlr/grammars-v4`)
- Para que a reescrita apenas acrescente o comentário, é necessário substituir o `skip` dos *tokens* que estão a ser desprezados, redireccionando-os para um canal escondido.
- Agora podemos criar um *listener* para resolver este problema.

Exemplo

```
import org.antlr.v4.runtime.*;

public class AddClassCommentListener extends Java8BaseListener {

   protected TokenStreamRewriter rewriter;

   public AddClassCommentListener(TokenStream tokens) {
      rewriter = new TokenStreamRewriter(tokens);
   }

   public void print() {
      System.out.print(rewriter.getText());
   }

   @Override public void enterNormalClassDeclaration(
         Java8Parser.NormalClassDeclarationContext ctx) {
      rewriter.insertBefore(ctx.start, "/**\n * class "+
                                       ctx.Identifier().getText()+
                                       "\n */\n");
   }
}
```

8.9 Desacoplar código da gramática - ParseTreeProperty

- Já vimos que podemos manipular a informação gerada na análise sintáctica de múltiplas formas:
  - Directamente na gramática recorrendo a acções e associando atributos a regras (argumentos, resultado, variáveis locais);
  - Utilizando *listeners*;
  - Utilizando *visitors*;
  - Associando atributos à gramática fazendo a sua manipulação dentro dos *listeners* e/ou *visitors*.
- Para associar informação extra à gramática, podemos acrescentar atributos à gramática (sintetizados, herdados ou variáveis locais às regras), ou utilizando os resultados dos métodos `visit`.
- Alternativamente, o `ANTLR4` fornece outra possibilidade: a sua biblioteca de *runtime* contém um *array* associativo que permite associar nós da árvore sintáctica com atributos – `ParseTreeProperty`.
- Vamos ver um exemplo com uma gramática para expressões aritméticas:

Exemplo

```
grammar Expr;

main: stat* EOF;

stat: expr;

expr: expr '*' expr # Mult
    | expr '+' expr # Add
    | INT           # Int
    ;

INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

Exemplo

```
import org.antlr.v4.runtime.tree.ParseTreeProperty;

public class ExprSolver extends ExprBaseListener {
   ParseTreeProperty<Integer> mapVal = new ParseTreeProperty<>();
   ParseTreeProperty<String> mapTxt = new ParseTreeProperty<>();

   public void exitStat(ExprParser.StatContext ctx) {
      System.out.println(mapTxt.get(ctx.expr()) + " = " +
                         mapVal.get(ctx.expr()));
   }

   public void exitAdd(ExprParser.AddContext ctx) {
      int left = mapVal.get(ctx.expr(0));
      int right = mapVal.get(ctx.expr(1));
      mapVal.put(ctx, left + right);
      mapTxt.put(ctx, ctx.getText());
   }

   public void exitMult(ExprParser.MultContext ctx) {
      int left = mapVal.get(ctx.expr(0));
      int right = mapVal.get(ctx.expr(1));
      mapVal.put(ctx, left * right);
      mapTxt.put(ctx, ctx.getText());
   }

   public void exitInt(ExprParser.IntContext ctx) {
      int val  = Integer.parseInt(ctx.INT().getText());
      mapVal.put(ctx, val);
      mapTxt.put(ctx, ctx.getText());
   }
}
```