Tema 1

Compiladores, Linguagens e Gramáticas

Introdução

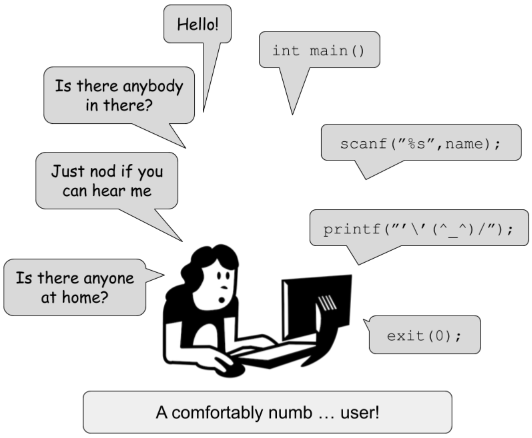
Compiladores, 2º semestre 2021-2022

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

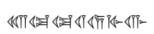
Conteúdo

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
}
```

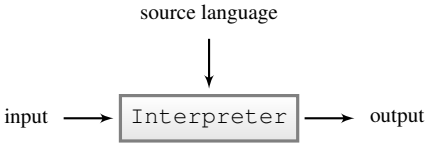
Exemplo 2: Calculadora

- 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, . . .).

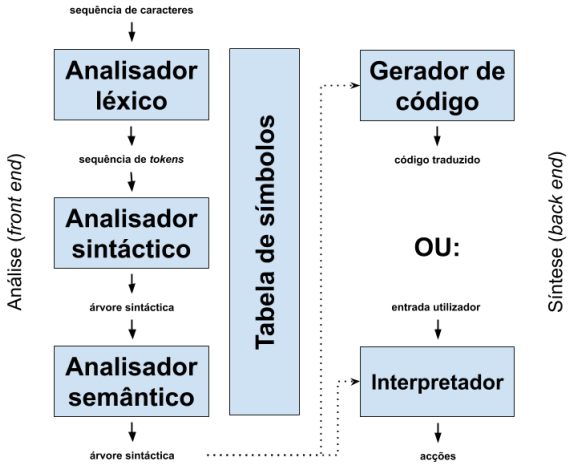
Exemplo: Calculadora

- 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:

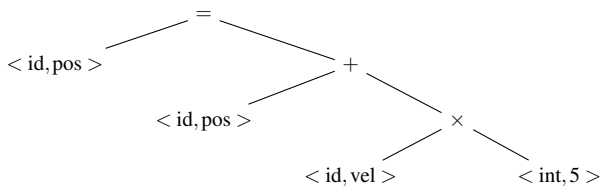
```
distance ( 0 , 0 ) ( 4 , 3 )
```

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

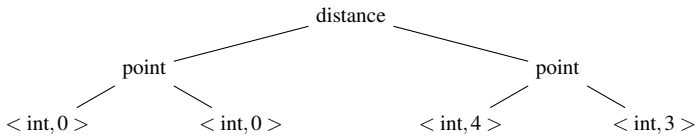
```
<distance> <(> <num,0> <,> <num,0> <)>
<(> <num,4> <,> <num,3> <)>
```

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

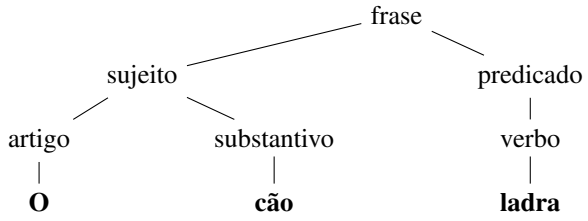
- Formalmente, a gramática anterior será:

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

- 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:

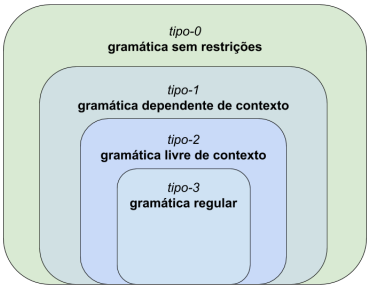
$$L_3 = \{u \mid u \in A^* \wedge u.\text{count}(1)\ mod\ 2 = 0\}$$

- 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).

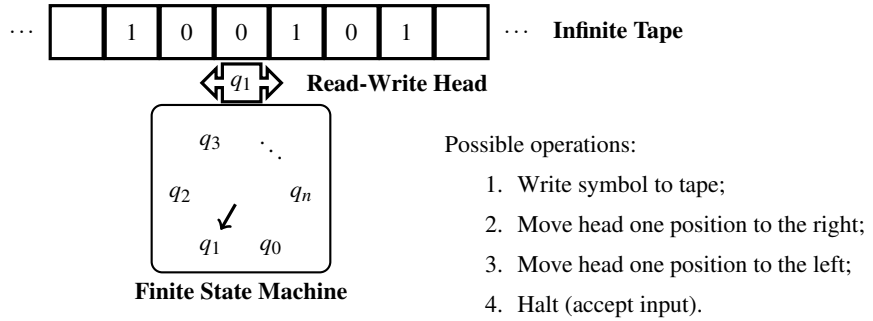


- Para cada um desses tipos podem ser definidos diferentes tipos de máquinas (algoritmos, autómatos) que as podem reconhecer.
- Quanto mais simples for a gramática, mas simples e eficiente é a máquina que reconhece essas linguagens.
- Cada classe de linguagens do *tipo-i* contém a classe de linguagens *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;

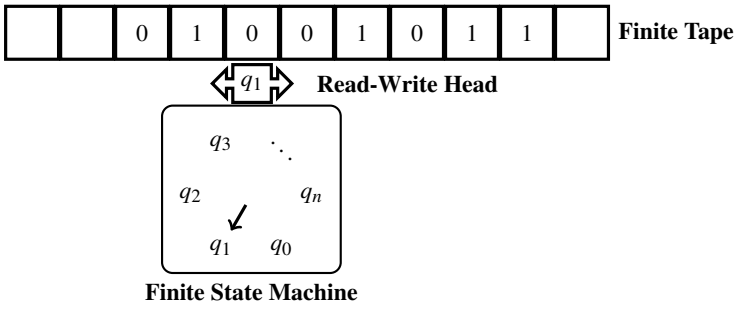
  h aceitar e terminar autómato.

- 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

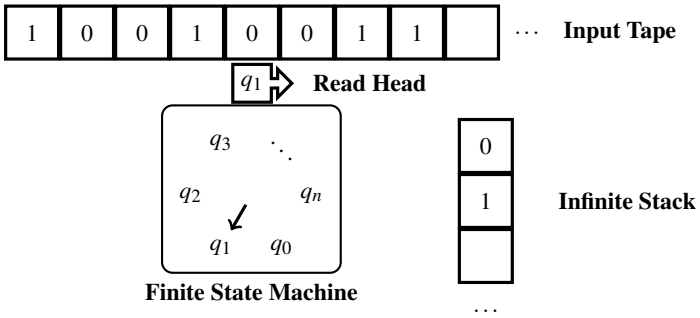
- $E_1\cdots\mathbf{0}111101110\cdots \rightarrow E_1\cdots0\mathbf{1}11101110\cdots \overset{*}{\rightarrow} E_2\cdots01111\mathbf{0}1110\cdots \rightarrow E_3\cdots0111\mathbf{1}1110\cdots \rightarrow E_3\cdots011111\mathbf{1}110\cdots \overset{*}{\rightarrow} E_3\cdots0111111\mathbf{1}0\cdots \rightarrow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

6.2.2 Autómatos linearmente limitados



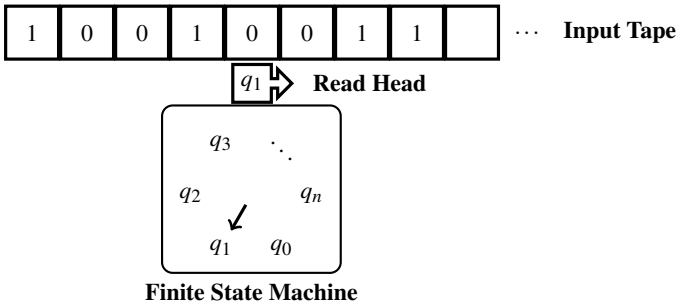
- Diferem das MT pela finitude da fita.

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`

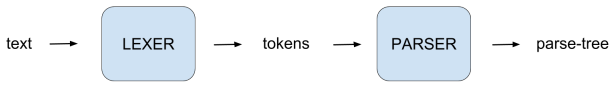
- Estes comandos estão disponíveis no *elearning* e fazem parte da instalação automática.

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)
```

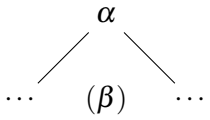
- As duas gramáticas – lexical e sintáctica – são expressas com instruções com a seguinte estrutura:

$$\alpha : \beta;$$

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`

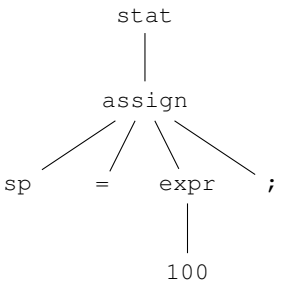
- Exemplo:

```
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:

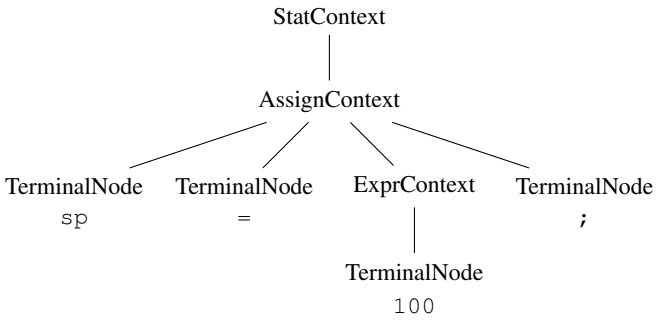
```
sp = 100;
```

- 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)

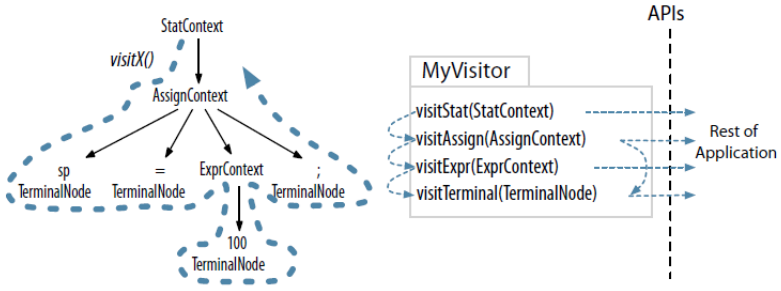
```
grammar Expr;                          → classes: ExprLexer and ExprParser
stat : assign ;                        → class StatContext in ExprParser
assign : ID '=' expr ';' ;             → class AssignContext in ExprParser
expr : INT ;                           → class ExprContext in ExprParser

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*

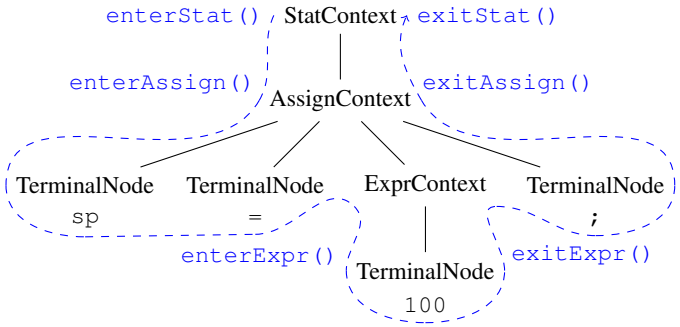
- Os objectos de contexto têm a si associada toda a informação relevante da análise sintáctica (*tokens*, referência aos nós filhos da árvore, etc.)
- Por exemplo o contexto `AssignContext` contém métodos `ID` e `expr` para aceder aos respectivos nós.

- 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.

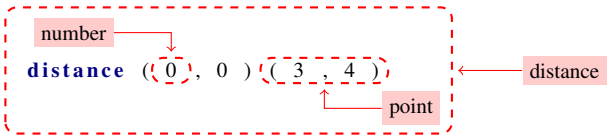
- A linguagem a ser executada na acção não tem de ser necessariamente Java (existem muitas outras possíveis, como C++ e python).
- Também podemos passar atributos para a regra (tipo passagem de argumentos para um método):

```
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.

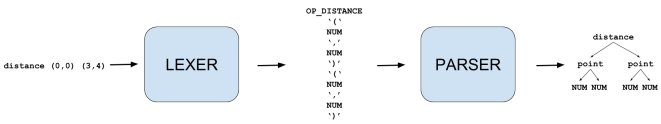
2.3 Exemplo figuras

- 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* ']'`

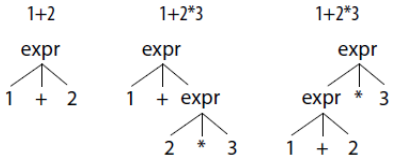
Sequence with terminator	`( instruction ';' )* // program sequence` `( row '\n' )*  // lines of data`
Sequence with separator	`expr ( ',' expr)*  // function call arguments` `( expr ( ',' expr)* )? // optional arguments`
Choice	`type: 'int' | 'float';` `instruction: conditional | loop | ... ;`

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:

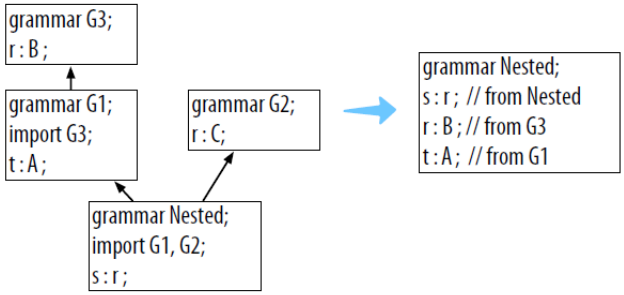
```
grammar ELang;
stat : (expr ';')* EOF ;
expr : INT ;
INT : [0-9]+ ;
WS : [ \r\t\n]+ -> skip ;
```

```
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

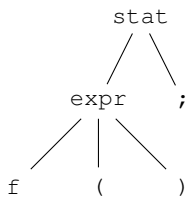
- Já vimos que nas regras sintácticas também pode haver ambiguidade.
- Os dois excertos seguintes exemplificam gramáticas ambíguas:

```
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*.

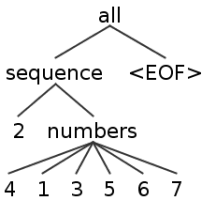
8.4 Predicados semânticos

- 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;
```

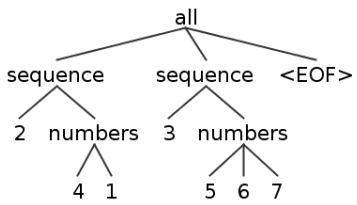
Com esta gramática, a árvore sintáctica gerada para a entrada `2 4 1 3 5 6 7` é:



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;
```

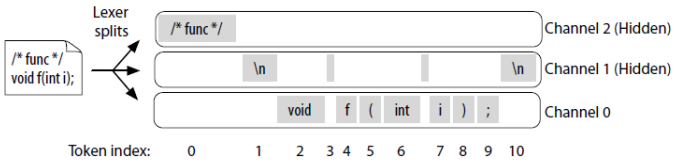
8.7 Enviar *tokens* para canais diferentes

- 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());
   }
}
```

Tema 3

Análise Semântica

Gramáticas de atributos, tabela de símbolos

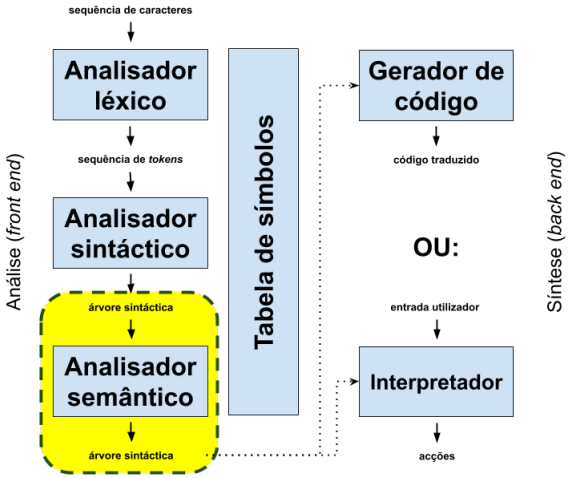
Compiladores, 2º semestre 2021-2022

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

Conteúdo

1 Análise semântica: Estrutura de um Compilador

- Vamos agora analisar com mais detalhe a fase de análise semântica:



- No processamento de uma linguagem a análise semântica deve garantir, tanto quanto possível, que o programa fonte faz sentido (mediante as regras definidas na linguagem).
- Erros semânticos comuns:
  - Variável/função não definida;
  - Tipos incompatíveis (e.g. atribuir número real a uma variável inteira, ou utilizar uma expressão não booleana na condições de uma instrução condicional);
  - Definir instrução num contexto errado (e.g. utilizar em `Java` a instrução `break` fora de um ciclo ou `switch`).
  - Aplicação sem sentido de instrução (e.g. importar uma *package* inexistente em `Java`).
- Em alguns casos, estes erros podem ser avaliados ainda durante a análise sintáctica; noutros casos, só é possível fazer essa verificação após uma análise sintáctica bem sucedida, fazendo uso da informação retirada dessa análise.

1.1 Avaliação dirigida pela sintaxe

- No processamento de linguagens, a avaliação semântica pode ser feita associando informação e acções às regras sintácticas da gramática (i.e. aos nós da *árvore sintáctica*).
- Este procedimento designa-se por *avaliação dirigida pela sintaxe*.
- Por exemplo, numa gramática para expressões aritméticas podemos associar aos nós da árvore uma variável com o tipo da expressão, e acções que permitam verificar a sua correcção (e não permitir, por exemplo, que se tente somar um booleano com um inteiro).
- Em ANTLR4, a associação de atributos e acções à árvore sintáctica, pode ser feita durante a própria análise sintáctica, e/ou posteriormente recorrendo a *visitors* e/ou *listeners*.

1.2 Detecção estática ou dinâmica

- A verificação de cada propriedade semântica de uma linguagem pode ser feita em dois tempos distintos:
  - Em *tempo dinâmico*: isto é, durante o *tempo de execução*;
  - Em *tempo estático*: isto é, durante o *tempo de compilação*.
- Só em compiladores fazem sentido verificações estáticas de propriedades semânticas.

- Em interpretadores as fases de análise e síntese da linguagem são ambas feitas em tempo de execução, pelo que as verificações são sempre dinâmicas.
- A verificação estática tem a vantagem de garantir, em tempo de execução, que certos erros nunca vão ocorrer (dispensando a necessidade de proceder à sua depuração e teste).

2 Sistema de tipos

- O sistema de tipos de uma linguagem de programação é um sistema lógico formal, com um conjunto de regras semânticas, que por associação de uma propriedade (tipo) a entidades da linguagem (expressões, variáveis, métodos, etc.) permite a detecção de uma classe importante de erros semânticos: *erros de tipos*.
- A verificação de erros de tipo, é aplicável nas seguintes operações:
  - Atribuição de valor: $v = e$
  - Aplicação de operadores: $e_1 + e_2$ (por exemplo)
  - Invocação de funções: $f(a)$
  - Utilização de classes/estruturas: $o.m(a)$ ou $data.field$
- Outras operações, como por exemplo a utilização arrays, podem também envolver verificações de tipo. No entanto, podemos considerar que as operações sobre arrays são atribuições de valor e aplicação de métodos especiais.
- Diz-se que qualquer uma destas operações é válida quando existe *conformidade* entre as propriedades de tipo das entidades envolvidas.
- A conformidade indica se um tipo T_1 pode ser usado onde se espera um tipo T_2. É o que acontece quando $T_1 = T_2$.
  - Atribuição de valor ($v = e$).
    O tipo de e tem de ser conforme com o tipo de v
  - Aplicação de operadores ($e_1 + e_2$).
    Existe um operador $+$ aplicável aos tipos de e_1 e e_2
  - Invocação de funções ($f(a)$).
    Existe uma função global f que aceita argumentos a conformes com os argumentos formais declarados dessa função.
  - Utilização de classes/estruturas ($o.m(a)$ ou $data.field$).
    Existe um método m na classe correspondente ao objecto o, que aceita argumentos a conformes com os argumentos formais declarados desse método; e existe um campo $field$ na estrutura/classe de $data$.

3 Gramáticas de atributos

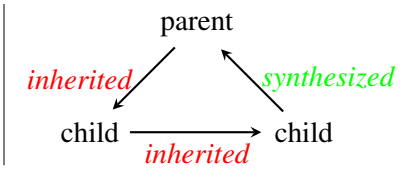
- Já vimos que atribuir sentido ao código fonte de uma linguagem requer, não só, correcção sintáctica (assegurada por gramáticas independentes de contexto) como também correcção semântica.
- Nesse sentido, é de toda a conveniência ter acesso a toda a informação gerada pela análise sintáctica, i.e. à árvore sintáctica, e poder associar nova informação aos respectivos nós.
- Este é o objectivo da *gramática de atributos*:
  - Cada símbolo da gramática da linguagem (terminal ou não terminal) pode ter a si associado um conjunto de zero ao mais *atributos*.
  - Um atributo pode ser um número, uma palavra, um tipo, ...
  - O cálculo de cada atributo tem de ser feito tendo em consideração a dependência da informação necessária para o seu valor.

- Entre os diferentes tipos de atributos, existem alguns cujo valor depende apenas da sua vizinhança sintáctica.
  - Um desses exemplos é o valor de uma expressão aritmética (que para além disso, depende apenas do próprio nó e, eventualmente, de nós descendentes).
- Existem também atributos que (podem) depender de informação remota.
  - É o caso, por exemplo, do tipo de dados de uma expressão que envolva a utilização de uma variável ou invocação de um método.

3.1 Dependência local: classificação de atributos

- Os atributos podem ser classificados duas formas, consoante as dependências que lhes são aplicáveis:
  1. Dizem-se *sintetizados*, se o seu valor depende apenas de nós descendentes (i.e. se o seu valor depende apenas dos símbolos existentes no respectivo corpo da produção).
  2. Dizem-se *herdados*, se depende de nós "irmãos"ou de nós ascendentes.

parent → child child



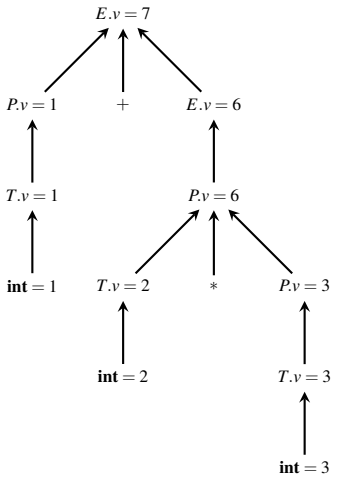
- Formalmente podem-se designar os atributos anotando com uma seta no sentido da dependência (para cima, nos atributos sintetizados; e para baixo nos herdados).

Exemplo dependência local: expressão aritmética

- Considere a seguinte gramática:

$$\begin{aligned} E &\rightarrow P + E \mid P \\ P &\rightarrow T * P \mid T \\ T &\rightarrow (E) \mid \textbf{int} \end{aligned}$$

- Se quisermos definir um atributo v para o valor da expressão, temos um exemplo de um atributo sintetizado.
- Por exemplo, para a entrada — $1+2*3$ — temos a seguinte árvore sintáctica anotada:



$$\begin{aligned} E &\rightarrow P + E \mid P \\ P &\rightarrow T * P \mid T \\ T &\rightarrow (E) \mid \textbf{int} \end{aligned}$$

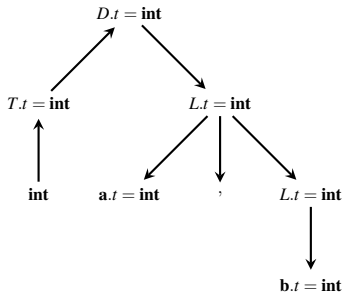
Produção	Regra semântica
$E_1 \rightarrow P + E_2$	$E_1.v = P.v + E_2.v$
$E \rightarrow P$	$E.v = P.v$
$P_1 \rightarrow T * P_2$	$P_1.v = T.v * P_2.v$
$P \rightarrow T$	$P.v = T.v$
$T \rightarrow (E)$	$T.v = E.v$
$T \rightarrow$ **int**	$T.v =$ **int**$.value$

Exemplo dependência local: declaração

- Considere a seguinte gramática:

$$\begin{aligned} D &\rightarrow T\,L \\ T &\rightarrow \textbf{int} \mid \textbf{real} \\ L &\rightarrow \textbf{id}, L \mid \textbf{id} \end{aligned}$$

- Se quisermos definir um atributo `t` para indicar o tipo de cada variável **id**, temos um exemplo de um atributo herdado.
- Por exemplo, para a entrada — **int** a, b — temos a seguinte árvore sintáctica anotada:



$$\begin{aligned} D &\rightarrow T\,L \\ T &\rightarrow \textbf{int} \mid \textbf{real} \\ L &\rightarrow \textbf{id}, L \mid \textbf{id} \end{aligned}$$

Produção	Regra semântica
$D \rightarrow T\,L$	$D.t = T.t$
	$L.t = T.t$
$T \rightarrow$ **int**	$T.t =$ **int**
$T \rightarrow$ **real**	$T.t =$ **real**
$L_1 \rightarrow$ **id**$, L_2$	**id**$.t = L_1.t$
	$L_2.t = L_1.t$
$L \rightarrow$ **id**	**id**$.t = L.t$

3.2 ANTLR4: Declaração de atributos associados à árvore sintáctica

- Podemos declarar atributos de formas distintas:
  1. Directamente na gramática independente de contexto recorrendo a argumentos e resultados de regras sintácticas;

```
expr[String type] returns[int value]: // type not used
      e1=expr '+' e2=expr
      {$value = $e1.value + $e2.value;}        #Add
```

```
    |  INT
       {$value = Integer.parseInt($INT.text);} #Int
    ;
```

2. Indirectamente fazendo uso do *array* associativo `ParseTreeProperty`:

```
protected ParseTreeProperty<Integer> value =
   new ParseTreeProperty<>();
...
@Override public void exitInt(ExprParser.IntContext ctx){
   value.put(ctx, Integer.parseInt(ctx.INT().getText()));
}
...
@Override public void exitAdd(ExprParser.AddContext ctx){
   int left = value.get(ctx.e1);
   int right = value.get(ctx.e2);
   value.put(ctx, left + right);
}
```

- Podemos ainda utilizar o resultados dos métodos `visit`.

ANTLR4: Declaração de atributos `ParseTreeProperty`

- Este *array* tem como chave nós da árvore sintáctica, e permite simular quer argumentos, quer resultados, de regras.
- A diferença está nos locais onde o seu valor é atribuído e acedido.
- Para simular a passagem de *argumentos* basta atribuir-lhe o valor *antes* de percorrer o respectivo nó (nos *listeners* usualmente nos métodos `enter..`), sendo o acesso feito no *próprio* nó.
- Para simular *resultados*, faz-se como no exemplo dado (i.e. atribui-se o valor no *próprio* nó, e acede-se nos nós *ascendentes*).

Gramáticas de atributos em ANTLR4: síntese

- Podemos associar três tipos de informação a regras sintácticas:
  1. Informação com origem em regras utilizadas no corpo da regra (atributos sintetizados);
  2. Informação com origem em regras que utilizam esta regra no seu corpo (atributos herdados);
  3. Informação local à regra.
- Em `ANTLR4` a utilização directa de todos estes tipos de atributos é muito simples e intuitiva:
  1. Atributos sintetizados: resultado de regras;
  2. Atributos herdados: argumentos de regras;
  3. Atributos locais.
- Alternativamente, podemos utilizar o *array* associativo `ParseTreeProperty` (que se justifica apenas para as duas primeiras, já que para a terceira podemos utilizar variáveis locais ao método respectivo); ou o resultado dos métodos `visit` (no caso de se utilizar *visitors*) para atributos sintetizados.

4 Tabela de símbolos

- A gramática de atributos é adequada para lidar com atributos com dependência local.
- No entanto, podemos ter informação cuja origem não tem dependência directa na árvore sintáctica (por exemplo, múltiplas aparições duma variável), ou que pode mesmo residir no processamento de outro código fonte (por exemplo, nomes de classes definidas noutro ficheiro).
- Assim, sempre que a linguagem utiliza símbolos para representar entidades do programa – como sejam: variáveis, funções, registos, classes, etc. – torna-se necessário associar à identificação do símbolo (geralmente um identificador) a sua definição (categoria do símbolo, tipo de dados associado).

- É para esse fim que existe a *tabela de símbolos*.
- A tabela de símbolos é um *array* associativo, em que a chave é o nome do símbolo, e o elemento um objecto que define o símbolo.
- As tabelas de símbolos podem ter um alcance global, ou local (por exemplo: a uma bloco de código ou a uma função).

- A informação associada a cada símbolo depende do tipo de linguagem definida, assim como de estarmos na presença de um interpretador ou de um compilador.
- São exemplos dessas propriedades:
  - **Nome**: nome do símbolo (chave do *array* associativo);
  - **Categoria**: o que é que o símbolo representa, classe, método, variável de objecto, variável local, etc.;
  - **Tipo**: tipo de dados do símbolo;
  - **Valor**: valor associado ao símbolo (apenas no caso de interpretadores).
  - **Visibilidade**: restrição no acesso ao símbolo (para linguagens com encapsulamento).

Tabela de símbolos: implementação

- Numa aproximação orientada por objectos podemos definir a classe abstracta `Symbol`:

```
public abstract class Symbol {
   public Symbol(String name, Type type) { ... }
   public String name() { ... }
   public Type type() { ... }
}
```

- Podemos agora definir uma variável:

```
public class VariableSymbol extends Symbol
{
   public VariableSymbol(String name, Type type) {
      super(name, type);
   }
}
```

- A classe `Type` permite a identificação e verificação da conformidade entre tipos:

```
public abstract class Type {
   protected Type(String name) { ... }
   public String name() { ... }
   public boolean subtype(Type other) {
      assert other != null;
      return name.equals(other.name());
   }
}
```

- Podemos agora implementar tipos específicos:

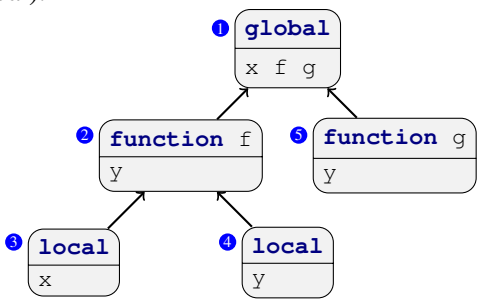
```
public class RealType extends Type {
   public RealType() { super("real"); }
}


public class IntegerType extends Type {
   public IntegerType() { super("integer"); }

   public boolean subtype(Type other) {
      return super.subtype(other) ||
             other.name().equals("real");
   }
}
```

4.1 Agrupando símbolos em contextos

- Se a linguagem é simples, contendo um único contexto de definição de símbolos, então o tempo de vida dos símbolos está ligado ao tempo de vida do programa, sendo suficiente uma única tabela de símbolos.
- No entanto, se tivermos a possibilidade de definir símbolos em contextos diferentes, então precisamos de resolver o problema dos símbolos terem tempos de vida (e/ou visibilidade) que dependem do contexto dentro do programa.
- Considere como exemplo o seguinte código (na linguagem `C`):

```
❶ // start of global scope
  int x;          // define variable x in global scope
❷ void f() {      // define function f in global scope
    int y;        // define variable y in local scope of f
❸   { int x; }   // define variable x in nested local scope
❹   { int y; }   // define variable y in another nested local scope
  }
❺ void g() {      // define function g in global scope
    int y;        // define variable y in local scope of g
  }
...
```

- A numeração identifica os diferentes contextos de símbolos.
- Um aspecto muito importante é o facto dos contextos poderem ser definidos dentro de outros contextos.
- Assim o contexto ❷ está definido dentro do contexto ❶; e, por sua vez, o contexto ❸ está definido dentro do ❷.
- Em ❹ o símbolo `x` está definido em ❶.
- Para representar adequadamente esta informação estrutura-se as diferentes tabelas de símbolos numa árvore onde cada nó representa uma pilha de tabelas de símbolos a começar nesse nó até à raiz (tabela de símbolos global).



- Consoante o ponto onde estamos no programa. temos uma pilha de tabelas de símbolos definida para resolver os símbolos.
- Pode haver repetição de nomes de símbolos, valendo o definido na tabela mais próxima (no ordem da pilha).
- Caso seja necessário percorrer a árvore sintáctica várias vezes, podemos registar numa lista ligada a sequência de pilhas de tabelas de símbolos que são aplicáveis em cada ponto do programa.

5 Instruções restringidas por contexto

- Algumas linguagens de programação restringem a utilização de certas instruções a determinados contexto.
- Por exemplo, em Java as instruções `break` e `continue` só podem ser utilizadas dentro de ciclos ou da instrução condicional `switch`.
- A verificação semântica desta condição é muito simples de implementar, podendo ser feita durante a análise sintáctica recorrendo a predicados semânticos e um contador (ou uma pilha) que registe o contexto.

```
@parser::members {
   int acceptBreak=0;
}
...
forLoop: 'for' '(' expr ';' expr ';' expr ')'
         {acceptBreak++;}
         instruction
         {acceptBreak--;}
       ;
break: {acceptBreak > 0}? 'break' ';'
     ;
instruction: forLoop | break | ...
           ;
```

6 ANTLR4: gestão de erros

6.1 ANTLR4: relatar erros

- Por omissão o ANTLR4 faz uma gestão de erros automática, que, em geral, responde bem às necessidades.
- No entanto, por vezes é necessário ter algum controlo sobre este processo.
- No que diz respeito à apresentação de erros, por omissão o ANTLR4 formata e envia essa informação para a saída *standard* da consola.
- Esse comportamento pode ser redefinido com a interface `ANTLRErrorListener`.
- Como o nome indica, o padrão de software utilizado é o de um *listener*, e tal como nos temos habituado em ANTLR existe uma classe base (com os métodos todos implementados sem código): `BaseErrorListener`
- O método `syntaxError` é invocado pelo ANTLR na presença de erros e aplica-se ao analisador sintáctico.

Relatar erros: exemplo 1

- Como exemplo podemos definir um *listener* que escreva também a pilha de regras do parser que estão activas.

```
import org.antlr.v4.runtime.*;
import java.util.List;
import java.util.Collections;

public class VerboseErrorListener extends BaseErrorListener {
   @Override public void syntaxError(Recognizer<?, ?> recognizer,
         Object offendingSymbol,
         int line, int charPositionInLine,
         String msg,
         RecognitionException e)
   {
      Parser p = ((Parser)recognizer);
      List<String> stack = p.getRuleInvocationStack();
      Collections.reverse(stack);
      System.err.println("rule stack: "+stack);
      System.err.println("line "+line+":"+charPositionInLine+
            " at "+offendingSymbol+": "+msg);
   }
}
```

- Podemos agora desactivar os *listeners* definidos por omissão e activar o novo *listener*:

```
...
AParser parser = new AParser(tokens);
parser.removeErrorListeners(); // remove ConsoleErrorListener
parser.addErrorListener(new VerboseErrorListener()); // add ours
parser.mainRule(); // parse as usual
...
```

- Note que podemos detectar a existência de erros após a análise sintáctica (já feito pelo `antlr4-main`):

```
...
parser.mainRule(); // parse as usual
if (parser.getNumberOfSyntaxErrors() > 0) {
   ...
}
```

- Podemos também passar todos os erros de reconhecimento de *tokens* para a análise sintáctica:

```
grammar AParser;
...
/**
Last rule in grammar to ensure all errors are passed to the parser
 */
ERROR: . ;
```

Relatar erros: exemplo 2

- Outro *listener* que escreva os erros numa janela gráfica:

```
import org.antlr.v4.runtime.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;

public class DialogErrorListener extends BaseErrorListener {
   @Override public void syntaxError(Recognizer<?, ?> recognizer,
         Object offendingSymbol, int line, int charPositionInLine,
         String msg, RecognitionException e)
   {
      Parser p = ((Parser)recognizer);
      List<String> stack = p.getRuleInvocationStack();
      Collections.reverse(stack);
      StringBuilder buf = new StringBuilder();
      buf.append("rule stack: "+stack+" ");
      buf.append("line "+line+":"+charPositionInLine+" at "+
            offendingSymbol+": "+msg);
      JDialog dialog = new JDialog();
      Container contentPane = dialog.getContentPane();
      contentPane.add(new JLabel(buf.toString()));
      contentPane.setBackground(Color.white);
      dialog.setTitle("Syntax error");
      dialog.pack();
      dialog.setLocationRelativeTo(null);
      dialog.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
      dialog.setVisible(true);
   }
}
```

6.2 ANTLR4: recuperar de erros

- A recuperação de erros é a operação que permite que o analisador sintáctico continue a processar a entrada depois de detectar um erro, por forma a se poder detectar mais do que um erro em cada compilação.
- Por omissão o ANTLR4 faz uma recuperação automática de erros que funciona razoavelmente bem.
- As estratégias seguidas pela ANTLR4 para esse fim são as seguintes:
  - inserção de *token*;
  - remoção de *token*;
  - ignorar *tokens* até sincronizar novamente a gramática com o fim da regra actual.
- (Não vamos detalhar mais este ponto.)

6.3 ANTLR4: alterar estratégia de gestão de erros

- Por omissão a estratégia de gestão de erros do ANTLR4 tenta recuperar a análise sintáctica utilizando uma combinação das estratégias atrás sumariamente apresentadas.
- A interface `ANTLRErrorStrategy` permite a definição de novas estratégias, existindo duas implementações na biblioteca de suporte: `DefaultErrorStrategy` e `BailErrorStrategy`.
- A estratégia definida em `BailErrorStrategy` assenta na terminação imediata da análise sintáctica quando surge o primeiro erro.
- A documentação sobre como lidar com este problema pode ser encontrada na classe `Parser`.
- Para definir uma nova estratégia de gestão de erros utiliza-se o seguinte código:

```
...
AParser parser = new AParser(tokens);
parser.setErrorHandler(new BailErrorStrategy());
...
```

Tema 4

Síntese

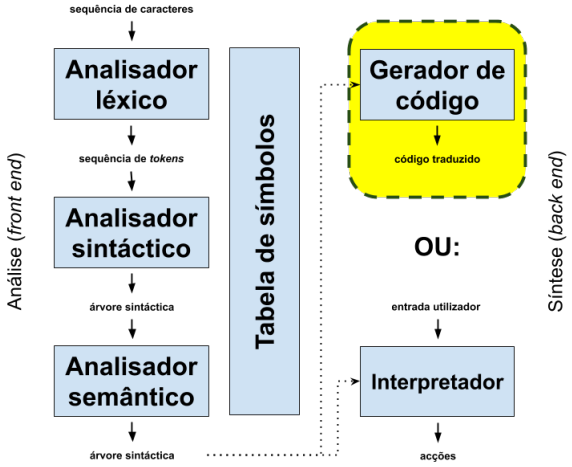
Geração de código e gestão de erros

Compiladores, 2º semestre 2021-2022

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

Conteúdo

1 Síntese: geração de código



- Podemos definir o objectivo de um compilador como sendo *traduzir* o código fonte de uma linguagem para outra linguagem.

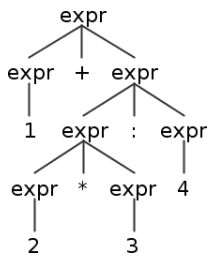
source language → `Compiler` → target language

- A geração do código para a linguagem destino pode ser feita por diferentes fases (podendo incluir fases de optimização), mas nós iremos abordar apenas uma única fase.
- A estratégia geral consiste em identificar *padrões de geração de código*, e após a análise semântica percorrer novamente a árvore sintáctica (mas já com a garantia muito importante de inexistência de erros sintácticos e semânticos) gerando o código destino nos pontos apropriados.

Exemplo: 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);
   }
}
```

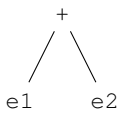
1.1 Geração de código máquina

- Tradicionalmente, o ensino de processadores de linguagens tende a dar primazia à geração de código baixo nível (linguagem máquina, ou *assembly*).

- A larga maioria da bibliografia mantém esse enfoque.
- No entanto, do ponto de vista prático serão poucos os programadores que, fazendo uso de ferramentas para gerar processadores de linguagens, necessitam ou ambicionam este tipo de geração de código.
- Nesta disciplina vamos, alternativamente, discutir a geração de código numa perspectiva mais abrangente, incluindo a geração de código em linguagens de alto nível.
- No que diz respeito à geração de código em linguagens de baixo nível, é necessário um conhecimento robusto em arquitectura de computadores e lidar com os seguintes aspectos:
  - Representação e formato da informação (formato para números inteiros, reais, estruturas, *array*, etc.);
  - Gestão e endereçamento de memória;
  - Implementação de funções (passagem de argumentos e resultado, suporte para recursividade com pilha de chamadas e *frame pointers*);
  - Alocação de registos do processador.
- (Consultar a bibliografia recomendada para estudar este tipo de geração de código.)

1.2 Geração de código

- Seja qual for o nível da linguagem destino, uma possível estratégia para resolver este problema consiste em identificar sem ambiguidade *padrões de geração de código* associados a cada *elemento gramatical da linguagem*.
- Para esse fim, é necessário definir o contexto de geração de código para cada elemento (por exemplo, geração de instruções na linguagem destino, ou atribuir a valor a uma variável), e depois garantir que o mesmo é compatível com todas as utilizações do elemento.



$$\begin{array}{l} \cdots(e_1) \\ v_1 = e_1 \\ \cdots(e_2) \\ v_1 = e_2 \\ \\ v_+ = v_1 + v_2 \end{array}$$

- Como a larguíssima maioria das linguagens destino são textuais, esses padrões de geração de código consistem em padrões de geração de texto.
- Assim sendo, em `Java`, poderíamos delegar esse problema no tipo de dados `String`, `StringBuilder`, ou mesmo na escrita directa de texto em em ficheiro (ou no *standard output*).
- No entanto, também aí o ambiente `ANTLR4` fornece uma ajuda mais estruturada, sistemática e modular para lidar com esse problema.

2 *String Template*

- A biblioteca *String Template* fornece uma solução estruturada para a geração de código textual.
- O software e documentação podem ser encontrados em `http://www.stringtemplate.org`
- Para ser utilizada é apenas necessário o pacote `ST-4.?.jar` (a instalação feita do antlr4 já incluiu este pacote).
- Vejamos um exemplo simples:

```
import org.stringtemplate.v4.*;
...
// code gen. pattern definition with <name> hole:
ST hello = new ST("Hello, <name>");
// hole pattern definition:
hello.add("name", "World");
// code generation (to standard output):
System.out.println(hello.render());
```

- Mesmo sendo um exemplo muito simples, podemos já verificar que a definição do padrão de texto, está separada do preenchimento dos "buracos" (atributos ou expressões) definidos, e da geração do texto final.
- Podemos assim delegar em partes diferentes do gerador de código, a definição dos padrões (que passam a pertencer ao contexto do elemento de código a gerar), o preenchimento dos "buracos" definidos, e a geração do texto final de código.
- Os padrões são blocos de texto e expressões.
- O texto corresponde a código destino literal, e as expressões são em "buracos" que podem ser preenchidos com o texto que se quiser.
- Sintaticamente, as expressões são identificadores delimitados por `<expr>` (ou por `$`).

```
import org.stringtemplate.v4.*;
...
ST assign = new ST("<var> = <expr>;\n");
assign.add("var", "i");
assign.add("expr", "10");
String output = assign.render();
System.out.println(output);
```

String Template Group

- Podemos também agrupar os padrões numa espécie de funções (módulo `STGroup`):

```
import org.stringtemplate.v4.*;
...
STGroup group = new STGroupString(
          "assign(var,expr) ::= \"<var> = <expr>;\""
      );
ST assign = group.getInstanceOf("assign");
assign.add("var", "i");
assign.add("expr", "10");
String output = assign.render();
System.out.println(output);
```

- Podemos também colocar cada função num ficheiro:

```
// file assign.st
assign(var,expr) ::= "<var> = <expr>;"
```

```
import org.stringtemplate.v4.*;
...
// assuming that assign.st is in current directory:
STGroup group = new STGroupDir(".");
ST assign = group.getInstanceOf("assign");
assign.add("var", "i");
assign.add("expr", "10");
String output = assign.render();
System.out.println(output);
```

- Uma melhor opção é optar por ficheiros modulares contendo grupos de funções/padrões:

```
// file templates.stg

templateName(arg1, arg2, ..., argN) ::= "single-line template"

templateName(arg1, arg2, ..., argN) ::= <<
multi-line template preserving indentation and newlines
>>

templateName(arg1, arg2, ..., argN) ::= <%
multi-line template that ignores indentation and newlines
%>
```

```
import org.stringtemplate.v4.*;
...
// assuming that templates.stg is in current directory:
STGroup allTemplates = new STGroupFile("templates.stg");
ST st = group.getInstanceOf("templateName");
...
```

String Template: dicionários e condicionais

- Neste módulos podemos ainda definir dicionários (arrays associativos).

```
typeValue ::= [
    "integer":"int",
    "real":"double",
    "boolean":"boolean",
    default:"void"
]
```

- Na definição de padrões podemos utilizar uma instrução condicional que só aplica o padrão caso o atributo seja adicionado:

```
stats(stat) ::= <<
<if(stat)><stat; separator="\n"><endif>
>>
```

- O campo `separator` indica que em em cada operação `add` em `stat`, se irá utilizar esse separador (no caso, uma mudança de linha).

String Template: Funções

- Podemos ainda definir padrões utilizando outros padrões (como se fossem funções).

```
module(name,stat) ::= <<
public class <name> {
    public static void main(String[] args) {
        <stats(stat)>
    }
}
>>

conditional(stat,var,stat_true,stat_false) ::= <<
<stats(stat)>
if (<var>) {
    <stat_true>
}<if(stat_false)>
else {
    <stat_false>
}<endif>
>>
```

String Template: listas

- Também existe a possibilidade de utilizar listas para concatenar texto e argumentos de padrões:

```
binaryExpression(type,var,e1,op,e2) ::=
    "<decl(type,var,[e1,\" \",op,\" \",e2])>"
```

- Para mais informação sobre as possibilidades desta biblioteca devem consultar a documentação existente em: `http://www.stringtemplate.org`.

2.1 Geração de código: padrões comuns

- Uma geração de código modular requer um contexto uniforme que permita a inclusão de qualquer combinação de código a ser gerado.
- Na sua forma mais simples, o padrão comum pode ser simplesmente uma sequência de instruções.

```
stats(stat) ::= <<
<if(stat)><stat; separator="\n"><endif>
>>

module(name,stat) ::= <<
public class <name>
{
    public static void main(String[] args)
    {
```

```
        <stats(stat)>
    }
}
>>
```

- Com este padrão, podemos inserir no lugar do "buraco" `stat` a sequência de instruções que quisermos.
- Naturalmente, que para uma geração de código mais complexa podemos considerar a inclusão de buracos para membros de classe, múltiplas classes, ou mesmo vários ficheiros.

- Para a linguagem `C`, teríamos o seguinte padrão para um módulo de compilação:

```
stats(stat) ::= <<
<if(stat)><stat; separator="\n"><endif>
>>

module(name,stat) ::= <<
#include <stdio.h>
#include <math.h>

int main()
{
    <stats(stat)>
}
>>
```

- Se a geração de código for guiada pela árvore sintáctica (como normalmente acontece), então os padrões de código a ser gerados devem ter em conta as definições gramaticais de cada símbolo, permitindo a sua aplicação modular em cada contexto.

2.2 Geração de código para expressões

- Para ilustrar a simplicidade e poder de abstração do *String Template* vamos estudar o problema de geração de código para expressões.
- Para resolver este problema de uma forma modular, podemos utilizar a seguinte estratégia:
    1. considerar que qualquer expressão tem a si associada uma variável (na linguagem destino) com o seu valor;
    2. para além dessa associação, podemos também associar a cada expressão um `ST` (`stats`) com as instruções que atribuem o valor adequado à variável.
- Como habitual, para fazer estas associações podemos definir atributos na gramática, fazer uso do resultados das funções de um *Visitor* ou utilizar a classe `ParseTreeProperty`
- Desta forma, podemos fácil e de uma forma modular, gerar código para qualquer tipo de expressão.

- Padrões para expressões (para `Java`) podem ser:

```
typeValue ::= [
    "integer":"int", "real":"double",
    "boolean":"boolean", default:"void"
]

init(value) ::= "<if(value)> = <value><endif>"
decl(type,var,value) ::=
    "<typeValue.(type)> <var><init(value)>;"

binaryExpression(type,var,e1,op,e2) ::=
    "<decl(type,var,[e1,\" \",op,\" \",e2])>"
```

- Para C apenas seria necessário mudar o padrão `typeValue`:

```
typeValue ::= [
    "integer":"int", "real":"double",
    "boolean":"int", default:"void"
]
```

3 Síntese: geração de código intermédio

3.1 Código de triplo endereço

- O padrão para expressões é um exemplo duma representação muito utilizada para geração de código baixo nível (em geral, intermédio, e não final), designada por *codificação de triplo endereço* (TAC).
- Esta designação tem origem nas instruções com a forma: $x = y$ **op** z
- No entanto, para além desta operação típica de expressões binárias, esta codificação contém outras instruções (ex: operações unárias e de controlo de fluxo).
- No máximo, cada instrução tem três operandos (i.e. três variáveis ou endereços de memória).
- Tipicamente, cada instrução TAC realiza uma operação elementar (e já com alguma proximidade com as linguagens de baixo nível dos sistemas computacionais).

3.2 TAC: Exemplo de expressões binárias

- Por exemplo a expressão $a + b * (c + d)$ pode ser transformada na sequência TAC:

```
t8 = d;
t7 = c;
t6 = t7+t8;
t5 = t6;
t4 = b;
t3 = t4*t5;
t2 = a;
t1 = t2+t3;
```

- Esta sequência – embora fazendo uso desregrado no número de registos (o que, num compilador gerador de código máquina, é resolvido numa fase posterior de optimização) – é codificável em linguagens de baixo nível.

3.3 TAC: Endereços e instruções

- Nesta codificação, um endereço pode ser:
  - Um nome do código fonte (variável, ou endereço de memória);
  - Uma constante (i.e. um valor literal);
  - Um nome temporário (variável , ou endereço de memória), criado na decomposição TAC.
- As instruções típicas do TAC são:
  1. Atribuições de valor de operação binária: $x = y$ **op** z
  2. Atribuições de valor de operação unária: $x =$ **op** y
  3. Instruções de cópia: $x = y$
  4. Saltos incondicionais e etiquetas: **goto** L e **label** L :
  5. Saltos condicionais: **if** x **goto** L ou **ifFalse** x **goto** L
  6. Saltos condicionais com operador relacional: **if** x **relop** y **goto** L (o operador pode ser de igualdade ou ordem)
  7. Invocações de procedimentos (**param** x_1 $\cdots$ **param** x_n ; **call** p,n ; $y =$ **call** p,n ; **return** y)
  8. Instruções com arrays (i.e. o operador é os parêntesis rectos, e um dos operandos é o índice inteiro).
  9. Instruções com ponteiros para memória (como em C)

3.4 Controlo de fluxo

- As instruções de controlo de fluxo são as instruções condicionais e os ciclos.
- Em linguagens de baixo nível muitas vezes estas instruções não existem.
- O que existe em alternativa é a possibilidade de dar "saltos" dentro do código recorrendo a endereços (*labels*) e a instruções de salto (*goto*, ...).

```
if (cond) {
   A;
}
else {
   B;
}
```

```
ifFalse cond goto l1
A
goto l2
label l1:
B
label l2:
```

- De forma similar podemos gerar código para ciclos:

```
while(cond) {
  A;
}
```

```
label l1:
ifFalse cond goto l2
A
goto l1
label l2:
```

3.5 Funções

- A geração de código para funções pode ser feita recorrendo a uma estratégia tipo "macro" (i.e. na invocação da funções é colocado o código que implementa a função), ou implementando módulos algorítmicos separados.
- Neste último caso (que, entre outras coisas, permite a recursividade), é necessária a definição de um bloco algorítmico separado, assim como implementar a passagem de argumentos/resultado para/de a função.
- A passagem de argumentos pode seguir diferentes estratégias: passagem por valor, passagem por referência de variáveis, passagem por referência de objectos/registos.
- Para termos implementações recursivas é necessário que se definam novas variáveis em cada invocação da função.
- A estrutura de dados que nos permite fazer isso de uma forma muito eficiente e simples é a pilha de execução.
- Esta pilha armazena os argumentos, variáveis locais à função e o resultado da função (permitindo ao código que invoca a função não só passar os argumentos à função como ir buscar o seu resultado).

- Geralmente as arquitecturas de linguagens de baixo nível (CPU's) têm instruções específicas para lidar com esta estrutura de dados.
- Vamos exemplificar esse procedimento:Este código apenas ilustra a ideia. Para uma análise mais detalhada devem consultar a temática de arquitectura de computadores *frame-pointer*.

```
// use:
··· f(x,y);
...
// define:
int f(int a, int b) {
   A;
   return r;
}
```

```
// use:
push 0 // result
push x
push y
call f,2
pop r // result
...
// define:
label f:
pop b
pop a
pop r
```

```
store stack-position
A
// reset stack to stack-position
restore stack-position
```

```
push r
return
```

Tema 3

Análise Lexical

Gramáticas regulares, autómatos finitos e expressões regulares

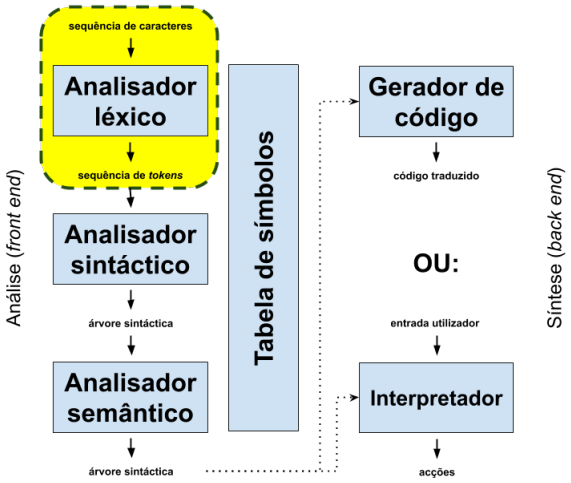
Compiladores+LFA, 2º semestre 2019-2020

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

Conteúdo

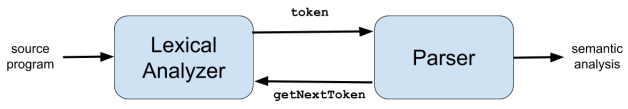
1 Análise Lexical: Estrutura de um Compilador

- O processo de compilação envolve diferentes fases:



- A primeira delas é a *análise léxica*, que consiste na conversão da sequência de caracteres de entrada numa sequência de elementos lexicais (*tokens*).

- A principal função da análise léxica é estruturar a sequência de carácteres da entrada numa sequência de *tokens* a serem processados pelo *parser*.

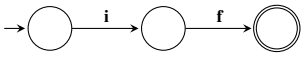


- No entanto, o analisador léxico efectua outras operações como sejam: a exclusão de espaços em branco e de comentários do *parser*, e a correlação entre erros (léxicos e sintácticos) com o código fonte (e.g. número da linha).

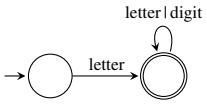
- A análise léxica pode ser feita recorrendo a gramáticas do tipo-3, ou seja, por *gramáticas regulares*, e a sua implementação computacional pode ser feita eficientemente recorrendo a *autómatos finitos*.

Análise Lexical: gramáticas regulares

- Uma *gramática é regular*, sse existir um *autómato finito* que a reconhece.
- Um autómato finito é uma *máquina de estados* com um estado inicial e um ou mais estados de aceitação (reconhecimento).
- As transições são feitas apenas tendo em conta o estado actual e a entrada.
- Um autómato finito para reconhecer a palavra reservada `if` será:



- Outro autómato finito que reconheça um identificador pode ser:



2 Linguagens regulares

- As gramáticas regulares geram *linguagens regulares*.
- A classe das linguagens regulares sobre um qualquer alfabeto A define-se indutivamente da seguinte forma:
  1. O conjunto vazio, $\emptyset$, é uma linguagem regular (LR).
  2. Qualquer que seja o símbolo $a \in A$, o conjunto $\{a\}$ é uma LR.
  3. Se L_1 e L_2 são linguagens regulares, então $L_1 \cup L_2$ (união) é uma LR.
  4. Se L_1 e L_2 são linguagens regulares, então $L_1 \cdot L_2$ (concatenação) é uma LR.
  5. Se L_1 é uma linguagem regular, então $(L_1)^*$ (fecho de Kleene) é uma LR.
  6. Nada mais é linguagem regular.
- Note que o conjunto $\{\varepsilon\}$, isto é, o conjunto composto pela palavra vazia, é também uma linguagem regular uma vez que: $\{\varepsilon\} = \emptyset^*$
- Uma vez que operações sobre LR geram uma LR, diz-se que a LR é *fechada* sobre as suas operações.

Linguagens regulares: exemplo 1

- Esta definição tem implicações interessantes.
- Uma delas é que qualquer linguagem finita, isto é que descreva um número finito de sequências de símbolos do seu alfabeto, é uma linguagem regular.
- Porquê?
  1. Seja $A = \{a_1, a_2, \ldots, a_n\}$ o alfabeto da linguagem L
  2. Então as linguagens $L_1 = \{a_1\}$, $L_2 = \{a_2\}$, $\ldots$, $L_n = \{a_n\}$ são LR (regra 2)
  3. Igualmente a linguagem $L_{\text{any}} = L_1 \cup L_2 \cup \cdots \cup L_n$ é também uma LR (regra 3)
  4. Qualquer que seja uma sequência finita de n símbolos do alfabeto A, podemos sempre descrevê-la como:
$$\text{seq}_n = \text{prefix}_{n-1}(\text{seq}_n) \cdot L_{\text{any}}$$
  5. Logo, a sequência será uma LR sse a subsequência $\text{prefix}_{n-1}(\text{seq}_n)$ também o for (regra 4).
  6. Aplicando indutivamente a demonstração, facilmente se chega à conclusão que se há-de de chegar à subsequência vazia, logo qualquer linguagem finita é uma linguagem regular.

Linguagens regulares: exemplo 1

- Uma vez que uma linguagem regular é uma linguagem reconhecida por um autómato finito é fácil, também por aí, chegarmos à mesma conclusão.
- Basta para tal considerar uma máquina de estados que implemente todas as transições das palavras da linguagem.
- Como o número de transições é finito (porque o número de palavras da linguagem também o é), então é sempre possível criar esse autómato finito.

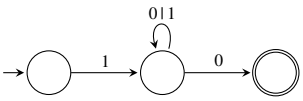
- Mantendo esta perspectiva, podemos ir mais longe e afirmar que numa linguagem finita, os autómatos que a reconhecem não têm ciclos.
- Da mesma forma, a existência de pelo menos um ciclo no autómato finito, implica necessariamente uma linguagem infinita[1]

Linguagens regulares: exemplo 2

- Mostre que o conjunto dos números binários começados em 1 e terminados em 0 é uma LR sobre o alfabeto $A = \{0,1\}$
- O conjunto pretendido pode ser representado por $L = \{1\} \cdot A^* \cdot \{0\}$

1. $\{1\}$ e $\{0\}$ são regulares (regra 2)
2. $A = \{0,1\} = \{0\} \cup \{1\}$ é regular (regra 3)
3. Se A é regular então A^* também é (regra 5)
4. Finalmente, $\{1\} \cdot A^* \cdot \{0\}$ é também regular (regra 4)

- Um autómato finito que reconhece esta linguagem pode ser:



- (Nota: A simples existência deste autómato também mostra a regularidade desta linguagem).

3 Gramáticas regulares

Definição de gramática

- Qualquer que seja a linguagem que se queira reconhecer, podemos sempre defini-las por intermédio de *gramáticas*.
- 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 $\alpha \to \beta$ 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 terminais e não terminais.

Definição de gramática regular

- Uma gramática diz-se *regular* (à direita) se para qualquer produção $(\alpha \to \beta \in P)$ as duas condições seguintes são satisfeitas:

  $\alpha \in N$

  $\beta \in T^* \cup T^* N$

- Alternativamente, podemos ter os não terminais (sempre) à esquerda:

  $\alpha \in N$

  $\beta \in T^* \cup N T^*$

[1] Esta constatação está de trás de um teorema muito importante para demostrar a não-regularidade de linguagens (*regular grammar pumping lemma*).

- Para linguagem: $\{\mathbf{1}\}\cdot T^*\cdot\{\mathbf{0}\}$ $(T=\{\mathbf{0},\mathbf{1}\})$, temos uma gramática regular à direita: $G=(T,\{S,X,Y\},S,P)$

$$\begin{aligned} S &\to \mathbf{1}X \\ X &\to Y \mid \mathbf{1}X \mid \mathbf{0}X \\ Y &\to \mathbf{0} \end{aligned}$$

- E no caso de uma gramática regular à esquerda:

$$\begin{aligned} S &\to X\,\mathbf{0} \\ X &\to Y \mid X\,\mathbf{1} \mid X\,\mathbf{0} \\ Y &\to \mathbf{1} \end{aligned}$$

- Uma gramática regular gera uma linguagem regular.
- As gramáticas regulares são também *fechadas* nas suas operações.
- Isto é, aplicar uma qualquer das operações definidas sobre gramáticas regulares resulta também numa gramática regular.

3.1 Operações sobre gramáticas regulares

Operações sobre gramáticas regulares: reunião

- Sejam $G_1=(T_1,N_1,S_1,P_1)$ e $G_2=(T_2,N_2,S_2,P_2)$ duas gramáticas regulares quaisquer com $N_1\cap N_2=\emptyset$
- A gramática $G=(T,N,S,P)$ onde:

$$\begin{aligned} T &= T_1\cup T_2 \\ N &= N_1\cup N_2\cup\{S\} \quad \text{com} \quad S\notin(N_1\cup N_2) \\ S &= S \\ P &= \{S\to S_1, S\to S_2\}\cup P_1\cup P_2 \end{aligned}$$

  é regular e gera a linguagem $L=L(G_1)\cup L(G_2)$
- A nova produção $S\to S_i$, com $i=1,2$, permite que G gere a linguagem $L(G_i)$

Operações sobre gramáticas regulares: reunião (exemplo)

- Sobre o conjunto de terminais $T=\{\mathrm{a},\mathrm{b},\mathrm{c}\}$, determine uma gramática regular que represente a linguagem

$$L=L_1\cup L_2$$

  sabendo que:

$$\begin{aligned} L_1 &= \{\mathrm{a}\,w : w\in T^*\} \\ L_2 &= \{w\,\mathrm{a} : w\in T^*\} \end{aligned}$$

- Vamos primeiro obter as GR que representam L_1 e L_2:

$$\begin{array}{lcl} S_1 \to \mathrm{a}X_1 & \qquad & S_2 \to \mathrm{a}S_2 \mid \mathrm{b}S_2 \mid \mathrm{c}S_2 \mid \mathrm{a} \\ X_1 \to \mathrm{a}X_1 \mid \mathrm{b}X_1 \mid \mathrm{c}X_1 \mid \varepsilon & & \end{array}$$

- Teremos assim como resultado a gramática:

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to \mathrm{a}X_1 \\ X_1 &\to \mathrm{a}X_1 \mid \mathrm{b}X_1 \mid \mathrm{c}X_1 \mid \varepsilon \\ S_2 &\to \mathrm{a}S_2 \mid \mathrm{b}S_2 \mid \mathrm{c}S_2 \mid \mathrm{a} \end{aligned}$$

Operações sobre gramáticas regulares: concatenação

- Sejam $G_1 = (T_1, N_1, S_1, P_1)$ e $G_2 = (T_2, N_2, S_2, P_2)$ duas gramáticas regulares quaisquer com $N_1 \cap N_2 = \emptyset$
- A gramática $G = (T, N, S, P)$ onde:

$$\begin{aligned} T &= T_1 \cup T_2 \\ N &= N_1 \cup N_2 \\ S &= S_1 \\ P &= \{A \to wS_2 : (A \to w) \in P_1 \ \wedge \ w \in T_1^*\} \cup \\ &\quad \{A \to w : (A \to w) \in P_1 \ \wedge \ w \in T_1^* N_1\} \cup \\ &\quad P_2 \end{aligned}$$

  é regular e gera a linguagem $L = L(G_1) \cdot L(G_2)$
- As produções da segunda gramática mantêm-se inalteradas.
- As produções da primeira gramática que terminam num não terminal, também se mantêm inalteradas.
- As produções da primeira gramática que só têm terminais ganham o símbolo inicial da segunda gramática no fim.

Operações sobre gramáticas regulares: concatenação (exemplo)

- Sobre o conjunto de terminais $T = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$, determine uma gramática regular que represente a linguagem

$$L = L_1 \cdot L_2$$

  sabendo que:

$$\begin{aligned} L_1 &= \{\mathrm{a}\, w : w \in T^*\} \\ L_2 &= \{w\, \mathrm{a} : w \in T^*\} \end{aligned}$$

- Recuperando as GR que representam L_1 e L_2:

$$\begin{array}{lcl} S_1 & \to & \mathrm{a}X_1 \\ X_1 & \to & \mathrm{a}X_1 \mid \mathrm{b}X_1 \mid \mathrm{c}X_1 \mid \varepsilon \end{array} \qquad \begin{array}{lcl} S_2 & \to & \mathrm{a}S_2 \mid \mathrm{b}S_2 \mid \mathrm{c}S_2 \mid \mathrm{a} \end{array}$$

- Teremos assim como resultado a gramática:

$$\begin{array}{lcl} S_1 & \to & \mathrm{a}X_1 \\ X_1 & \to & \mathrm{a}X_1 \mid \mathrm{b}X_1 \mid \mathrm{c}X_1 \mid S_2 \\ S_2 & \to & \mathrm{a}S_2 \mid \mathrm{b}S_2 \mid \mathrm{c}S_2 \mid \mathrm{a} \end{array}$$

Operações sobre gramáticas regulares: fecho de Kleene

- Seja $G_1 = (T_1, N_1, S_1, P_1)$ uma gramática regular qualquer.
- A gramática $G = (T, N, S, P)$ onde:

$$\begin{aligned} T &= T_1 \\ N &= N_1 \cup \{S\} \,\mathrm{com}\, S \notin N_1 \\ S &= S_1 \mid \varepsilon \\ P &= \{S \to S_1 \mid \varepsilon\} \cup \\ &\quad \{A \to wS : (A \to w) \in P_1 \wedge w \in T_1^*\} \cup \\ &\quad \{A \to w : (A \to w) \in P_1 \wedge w \in T_1^* N_1\} \end{aligned}$$

é regular e gera a linguagem $L = (L(G_1))^*$
- As produções que terminam num não terminal mantêm-se inalteradas
- As produções só têm terminais ganham o símbolo inicial no fim
- As novas produções $(S \to S_1 \mid \varepsilon)$ garantem que $(L(G_1))^n \subseteq L(G)$, para qualquer $n \geq 0$

Operações sobre gramáticas regulares: fecho de Kleene (exemplo)

- Sobre o conjunto de terminais $T = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$, determine uma gramática regular que represente a linguagem

$$L = L_1^*$$

sabendo que:

$$L_1 \quad = \quad \{\mathrm{a}\, w : w \in T^*\}$$

- Recuperando a GR que representa L_1:

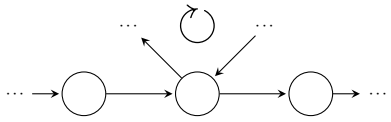
$$\begin{array}{lcl} S_1 & \to & \mathrm{a} X_1 \\ X_1 & \to & \mathrm{a} X_1 \mid \mathrm{b} X_1 \mid \mathrm{c} X_1 \mid \varepsilon \end{array}$$

- Teremos assim como resultado a gramática:

$$\begin{array}{lcl} S & \to & S_1 \mid \varepsilon \\ S_1 & \to & \mathrm{a} X_1 \\ X_1 & \to & \mathrm{a} X_1 \mid \mathrm{b} X_1 \mid \mathrm{c} X_1 \mid S \end{array}$$

3.2 Teorema da repetição para linguagens regulares

- Das várias definições equivalentes para linguagens e gramáticas regulares, talvez a mais fácil de compreender seja a de que uma linguagem será regular sse existir um autómato finito que a reconheça.
- Com esta definição é podemos com mais facilidade compreender que uma linguagem finita não só é sempre regular, como também o respectivo autómato não pode ter ciclos.
- Em sentido inverso, podemos também concluir que uma linguagem regular infinita tem de ter pelo menos um ciclo.
- Desta constatação podemos inferir que nas linguagens regulares infinitas, tem de ser sempre possível injectar (repetir) uma determinada sub-palavra as vezes que se quiser, mantendo a palavra resultante dentro da linguagem.
- Visualmente, considerando um autómato finito para este tipo de linguagens, essa palavra será a que resulta do ciclo que necessariamente tem de existir (em linguagens infinitas).



- Uma vez que o ciclo pode requerer palavras com uma dimensão mínima (suficiente para fechar o ciclo), esta condição (necessária mas não suficiente) requer um tamanho mínimo para palavras da linguagem.
- Por outro lado, nos autómatos finitos (para linguagens infinitas) podemos sempre identificar um primeiro ciclo, i.e. um ciclo que aparece a uma distância finita do inicio da palavra.

Teorema da repetição (*pumping lemma*) para linguagens regulares:: Numa linguagem regular infinita L, existe um tamanho p tal que para todas as palavras $w \in L$ para as quais: $|w| \geq p$, podemos particionar w em três palavras: $w = xyz$, tal que:
- $|y| > 0$ (padrão de repetição não vazio)
- $|xy| \leq p$ (primeiro ciclo a uma distância finita do início da palavra)
- $\forall k \geq 0 : xy^k z \in L$ (*pumping*)

- Note que esta é uma condição necessária para uma linguagem regular infinita, mas não suficiente.
- Isto é, podem existir linguagens não regulares onde esta condição se verifique.
- No entanto, a inexistência desta condição numa linguagem infinita é suficiente para determinar a sua não-regularidade.

- Temos assim que este teorema pode ser aplicado para *demostrar a não regularidade de linguagens*.
- Isto é, assumimos a regularidade da linguagem e vamos tentar demonstrar o oposto por contradição.
- Para demostrar a regularidade de uma linguagem podemos aplicar as condições que definem linguagens e gramáticas regulares (ver definição de linguagem regular na página 3).

Teorema da repetição para linguagens regulares: exemplo 1

- Determine a regularidade da linguagem $L_1 = \{a^i b^i : i \geq 0\}$.
- Em primeiro lugar note que não é possível aplicar a condição da concatenação, porque há uma dependência no número de repetições dos símbolos a e b. Isto é, muito embora quer a^i, quer b^i, $(i \geq 0)$ sejam regulares, o número de repetições i tem de ser o mesmo (condição não garantida pela concatenação).
- Dito de outra forma: $a^i b^i \neq a^* b^*$.
- Uma vez que a linguagem é infinita então, para ser regular, o teorema da repetição tem ser aplicável.
- Seja $w = a^n b^n$. Então $|w| = 2n$.
- Para particionar $w = xyz$ temos três possibilidades: y contém só sequências de a's, só sequências de b's, ou uma sequência com prefixos de a's e sufixos de b's.

- Os dois últimos casos quebram desde logo a condição $|xy| \leq p$ já que o x terá de conter a's e existem tantas palavras em L_1 quantas se queira com um número não limitado de a's em x.
- Resta o primeiro caso.
- Seja $x = a^p, y = a^q, z = a^r b^n$, então $p+q+r = n$ para que $w \in L_1$ e $q \neq 0$.
- Como $xy^2z \notin L_1$ (uma vez que $q+2*q+r \neq n$), então L_1 não é regular.

Teorema da repetição para linguagens regulares: exemplo 2

- Determine a regularidade da linguagem $L_2 = \{c^k a^i b^i : k \geq 0 \wedge i \geq 0\}$.
- Repare que neste caso o teorema da repetição aplica-se ($y = c$). No entanto seria estranho que esta linguagem fosse regular uma vez que contém a linguagem anterior (que não era regular).
- Não há aqui nenhuma contradição porque o teorema da repetido é uma condição necessária, mas não suficiente.
- Para determinar a regularidade neste caso temos de aplicar as operações aplicáveis a linguagens regulares.
- Assim: $L_2 = \{c^k : k \geq 0\} \cdot L_1$.
- A linguagem $\{c^k : k \geq 0\}$ é regular, mas L_1 não o é, pelo que L_2 também não vai ser.

4 Expressões regulares

- As expressões regulares foram introduzidas em 1956 por Stephen Kleene.
- O conjunto das expressões regulares sobre um alfabeto A define-se indutivamente da seguinte forma:

1. $()$ é uma expressão regular (ER) que representa a LR $\{\}$ ($\emptyset$).
2. Qualquer que seja o a $\in A$, a é uma ER que representa a LR $\{\text{a}\}$.
3. Se e_1 e e_2 são ER representando respectivamente as LR L_1 e L_2 , então $(e_1 \mid e_2)$ é uma ER representando a LR $L_1 \cup L_2$.

4. Se e_1 e e_2 são ER representando respectivamente as LR L_1 e L_2 , então (e_1e_2) é uma ER representando a LR $L_1 \cdot L_2$.
5. Se e_1 é uma ER representando a LR L_1 , então e_1^* é uma ER representando a LR $(L_1)^*$.
6. Nada mais é expressão regular.

- É habitual representar-se por $\{\varepsilon\}$ a ER $()^*$. Representa a linguagem $\{\varepsilon\}$.
- A evidente semelhança entre LR e ER não é casual. Ambas expressam *gramáticas regulares*.
- Uma expressão regular, tal como uma gramática regular, gera uma linguagem regular.
  - Logo, é possível converter uma gramática regular numa expressão regular que represente a mesma linguagem e *vice-versa*.
- Tal como as gramáticas regulares, as expressões regulares são *fechadas* nas suas operações.

Expressões regulares: exemplos

P: Determine uma ER que represente o conjunto de números binários começados por 1 e terminados por 0.
R: $1(0\,|\,1)^*0$

P: Determine uma ER que representa as sequências definidas sobre o alfabeto $A = \{\mathrm{a},\mathrm{b},\mathrm{c}\}$ que satisfazem o requisito de qualquer símbolo b ter um a imediatamente à sua esquerda e um c imediatamente à sua direita.
R: $(\mathrm{a}\,|\,\mathrm{abc}\,|\,\mathrm{c})^*$

P: Determine uma ER que represente as sequências binárias com um numero par de zeros.
R: $1^*(01^*01^*)^*$

Propriedades das expressões regulares

- Operação de escolha (|):
  - comutativa: $e_1\,|\,e_2 = e_2\,|\,e_1$
  - associativa: $e_1\,|\,(e_2\,|\,e_3) = (e_1\,|\,e_2)\,|\,e_3 = e_1\,|\,e_2\,|\,e_3$
  - existência de elemento neutro: $e_1\,|\,() = ()\,|\,e_1 = e_1$
  - idempotência: $e_1\,|\,e_1 = e_1$
- Operação de concatenação (implícita ou $\cdot$):
  - associativa: $e_1(e_2e_3) = (e_1e_2)e_3 = e_1e_2e_3$
  - existência de elemento neutro: $e_1\varepsilon = \varepsilon e_1 = e_1$
  - existência de elemento absorvente: $e_1() = ()e_1 = ()$ (a concatenação com o conjunto vazio resulta no próprio)
  - não goza da propriedade comutativa
- Operações de escolha e de concatenação:
  - concatenação distributiva relativamente à escolha:

    $e_1(e_2\,|\,e_3) = (e_1e_2)\,|\,(e_1e_3) = e_1e_2\,|\,e_1e_3$

    $(e_1\,|\,e_2)e_3 = (e_1e_3)\,|\,(e_2e_3) = e_1e_3\,|\,e_2e_3$
- Operação de fecho: $r^* = \varepsilon \mid r \mid rr \mid \ldots$

$(e^*)^* = e^*$

$(e_1^*\,|\,e_2^*)^* = (e_1\,|\,e_2)^*$

$(e_1\,|\,e_2)^* \neq e_1^*\,|\,e_2^*$

$(e_1e_2)^* \neq e_1^*e_2^*$

Simplificação notacional

- Para simplificar a escrita das expressões regulares (de forma análoga às expressões aritméticas) existe uma precedência bem definida na aplicação dos diferentes operadores.
- A ordem decrescente de precedência é a seguinte:
  1. Parêntesis
  2. Fecho de Kleene (*)
  3. Concatenação (implícita ou $\cdot$)
  4. Escolha ($|$)
- A utilização destas precedências permite simplificar as ER
  $e = ((((1^*)0)(1^*))0)(1^*) \Leftrightarrow e = 1^*01^*01^*$
  $e_1 \,|\, e_2 \cdot e_3^* = e_1 \,|\, (e_2 \cdot (e_3^*))$

Expressões regulares: exemplos

- Recuperando os exemplos anteriores.

P: Determine uma ER que represente o conjunto de números binários começados por 1 e terminados por 0.
R: $1(0\,|\,1)^*0 \Leftrightarrow (1((0\,|\,1)^*))0$

P: Determine uma ER que representa as sequências definidas sobre o alfabeto $A = \{\mathrm{a},\mathrm{b},\mathrm{c}\}$ que satisfazem o requisito de qualquer símbolo b ter um a imediatamente à sua esquerda e um c imediatamente à sua direita.
R: $(\mathrm{a}\,|\,\mathrm{abc}\,|\,\mathrm{c})^* \Leftrightarrow ((\mathrm{a}\,|\,((\mathrm{ab})\mathrm{c}))\,|\,\mathrm{c})^*$

P: Determine uma ER que represente as sequências binárias com um numero par de zeros.
R: $1^*(01^*01^*)^* \Leftrightarrow (1^*)(((((0(1^*))0)(1^*)))^*)$

Expressões regulares: mais exemplos

P: Sobre o alfabeto $A = \{0,1\}$ construa uma ER para a linguagem:

$$L = \{w : w \in A^* \wedge \#(0,w) = 2\}$$

R: $1^*01^*01^*$

P: Sobre o alfabeto $A = \{\mathrm{a},\mathrm{b},\cdots,\mathrm{z}\}$ construa uma ER para a linguagem:

$$L = \{w : w \in A^* \wedge \#(\mathrm{a},w) = 3\}$$

R: $(\mathrm{b}\,|\,\mathrm{c}\,|\,\cdots\,|\,\mathrm{z})^*\mathrm{a}\,(\mathrm{b}\,|\,\mathrm{c}\,|\,\cdots\,|\,\mathrm{z})^*\mathrm{a}\,(\mathrm{b}\,|\,\mathrm{c}\,|\,\cdots\,|\,\mathrm{z})^*\mathrm{a}\,(\mathrm{b}\,|\,\mathrm{c}\,|\,\cdots\,|\,\mathrm{z})^*$

Extensões notacionais

Por forma a simplificar ao máximo a construção de expressões regulares é usual definir-se algumas extensões.

- Uma ou mais ocorrências:
  $e^+ = ee^*$
- Uma ou nenhuma ocorrência:
  $e? = (e\,|\,\varepsilon)$
- Um símbolo dum sub-alfabeto:
  $[\mathrm{a}_1\mathrm{a}_2\ldots\mathrm{a}_n] = \mathrm{a}_1\,|\,\mathrm{a}_2\,|\ldots|\,\mathrm{a}_n$
  $[\mathrm{a}_1-\mathrm{a}_n] = \mathrm{a}_1\,|\,\mathrm{a}_2\,|\ldots|\,\mathrm{a}_n$
- Um símbolo fora dum sub-alfabeto:
  $[\hat{}\,\mathrm{a}_1\mathrm{a}_2\ldots\mathrm{a}_n]$ ou (`ANTLR`): $\sim[\mathrm{a}_1\mathrm{a}_2\ldots\mathrm{a}_n]$
  $[\hat{}\,\mathrm{a}_1-\mathrm{a}_n]$ ou (`ANTLR`): $\sim[\mathrm{a}_1-\mathrm{a}_n]$

– n ocorrências:
$e\{n\} = \underbrace{e{\cdot}e{\cdot}\ldots{\cdot}e}_{n}$
– de n_1 a n_2 ocorrências:
$e\{n1,n2\} = \underbrace{e{\cdot}e{\cdot}\ldots{\cdot}e}_{n_1,n_2}$

– n ou mais ocorrências:
$e\{n,\} = \underbrace{e{\cdot}e{\cdot}\ldots{\cdot}e}_{n,}$

Expressões regulares: mais exemplos

P: Sobre o alfabeto $A = \{0,1\}$ construa uma ER para a linguagem:

$$L = \{w : w \in A^* \wedge \#(0,w) = 2\}$$

R: $1^*01^*01^* = (1^*0)\{2\}1^*$

P: Sobre o alfabeto $A = \{\mathrm{a},\mathrm{b},\cdots,\mathrm{z}\}$ construa uma ER para a linguagem:

$$L = \{w : w \in A^* \wedge \#(\mathrm{a},w) = 3\}$$

R: $(\mathrm{b}|\mathrm{c}|\cdots|\mathrm{z})^*\mathrm{a}\,(\mathrm{b}|\mathrm{c}|\cdots|\mathrm{z})^*\mathrm{a}\,(\mathrm{b}|\mathrm{c}|\cdots|\mathrm{z})^*\mathrm{a}\,(\mathrm{b}|\mathrm{c}|\cdots|\mathrm{z})^*$
$= ([\mathrm{b}{-}\mathrm{z}]^*\mathrm{a})\{3\}[\mathrm{b}{-}\mathrm{z}]^*$

Outras extensões notacionais

Existem outras extensões a expressões regulares (utilizadas, por exemplo, em muitos comandos `UNIX`):

Símbolo:	Significado:
`.`	um símbolo qualquer diferente de `\n` (em `ANTLR` significa diferente de `EOF`)
`^`	palavra vazia no início de linha
`$`	palavra vazia no fim de linha
`\<`	palavra vazia no início de palavra
`\>`	palavra vazia no fim de palavra

4.1 Gramática para expressões regulares

- Podemos definir a linguagem das expressões regulares com uma gramática (A é o conjunto dos caracteres):

ER	$\rightarrow$	ER $'\|'$ Term	$\{alternativa\}$
ER	$\rightarrow$	Term	
Term	$\rightarrow$	Term Primary	$\{concatenação\}$
Term	$\rightarrow$	Primary	
Primary	$\rightarrow$	Factor $'*'$	$\{iteração\}$
Primary	$\rightarrow$	Factor	
Factor	$\rightarrow$	$'('$ ER $')'$	$\{grupo\}$
Factor	$\rightarrow$	A	$\{qualquer\ terminal\}$

- (Note que é uma gramática de tipo-2, i.e. independente de contexto.)

5 Conversão entre ER e GR

5.1 Conversão de ER para GR

- É suficiente obter a GR para as ER primitivas e aplicar as operações regulares sobre a GR
- A GR para a ER ε é dada por:
  $S \to \varepsilon$
- A GR para a ER a, qualquer que seja o a, é dada por:
  $S \to \mathrm{a}$
- Vamos exemplificar com a ER $e = (\mathrm{a}\,|\,\mathrm{b})^*\mathrm{a}$

1. Primeiro definimos regras para os símbolos terminais:

   $S_1 \to \mathrm{a}$

   $S_2 \to \mathrm{b}$
2. Para reconhecer $(\mathrm{a}\,|\,\mathrm{b})$ temos $(S = S_3)$:

   $S_3 \to S_1 \,|\, S_2$

   $S_1 \to \mathrm{a}$

   $S_2 \to \mathrm{b}$
3. Para reconhecer $(\mathrm{a}\,|\,\mathrm{b})^*$ temos $(S = S_3)$:

   $S_3 \to S_1 \,|\, S_2 \,|\, \varepsilon$

   $S_1 \to S_3\,\mathrm{a}$

   $S_2 \to S_3\,\mathrm{b}$
4. Por fim para reconhecer a ER $(\mathrm{a}\,|\,\mathrm{b})^*\mathrm{a}$ temos $(S = S_4)$:

   $S_4 \to S_3\,\mathrm{a}$

   $S_3 \to S_1 \,|\, S_2 \,|\, \varepsilon$

   $S_1 \to S_3\,\mathrm{a}$

   $S_2 \to S_3\,\mathrm{b}$

5.2 Conversão de GR para ER

- Seja $G_1 = (T_1, N_1, S_1, P_1)$ uma gramática regular qualquer.
- Uma ER que represente a mesma linguagem que a gramática G pode ser obtida por um processo de transformação de equivalência:
  1. Converte-se a gramática G no conjunto de triplos seguinte:

     $$\begin{aligned}\mathscr{E} \;=\; & \{E, \varepsilon, S\} \cup \\ & \{(A, w, B) : (A \to wB) \in P\} \cup \\ & \{(A, w, \varepsilon) : (A \to w) \in P\}\end{aligned}$$

     com $E \notin N \wedge w \in T^* \wedge A \in N \wedge B \in N$
  2. Removem-se, por transformações de equivalência, um a um, todos os símbolos de N, até se obter um único triplo da forma: (E, e, ε). A ER equivalente será a expressão e.
- Remoção dos símbolos de N:
  (A) Substituir todos os triplos da forma (A, β_i, B) por um único (A, w_1, B), onde $\mathrm{w}_1 = \beta_1 \,|\, \beta_2 \,|\, \cdots \,|\, \beta_n$
  (B) Substituir todos os triplos da forma (B, α_i, B) por um único (B, w_2, B), onde $\mathrm{w}_2 = \alpha_1 \,|\, \alpha_2 \,|\, \cdots \,|\, \alpha_m$
  (C) Substituir todos os triplos da forma (B, γ_i, C) por um único (B, w_3, C), onde $\mathrm{w}_3 = \gamma_1 \,|\, \gamma_2 \,|\, \cdots \,|\, \gamma_k$

(D) Substituir o triplo de triplos $((A, \mathrm{w}_1, B), (B, \mathrm{w}_2, B), (B, \mathrm{w}_3, C))$ pelo triplo $(A, \mathrm{w}_1 \mathrm{w}_2^* \mathrm{w}_3, C)$

- Vamos exemplificar com a seguinte GR:

$$
\begin{aligned}
S &\rightarrow aS \mid bS \mid cS \mid abaX \\
X &\rightarrow aX \mid bX \mid cX \mid \varepsilon \\
\mathscr{E} &= \{(E, \varepsilon, S), (S, \mathrm{a}, S), (S, \mathrm{b}, S), (S, \mathrm{c}, S), (S, \mathrm{aba}, X), \\
&\quad (X, \mathrm{a}, X), (X, \mathrm{b}, X), (X, \mathrm{c}, X), (X, \varepsilon, \varepsilon)\} \\
\text{(B)} &= \{(E, \varepsilon, S), (S, \mathrm{a}\mid\mathrm{b}\mid\mathrm{c}, S), (S, \mathrm{aba}, X), \\
&\quad (X, \mathrm{a}, X), (X, \mathrm{b}, X), (X, \mathrm{c}, X), (X, \varepsilon, \varepsilon)\} \\
\text{(D)} &= \{(E, (\mathrm{a}\mid\mathrm{b}\mid\mathrm{c})^*\mathrm{aba}, X), \\
&\quad (X, \mathrm{a}, X), (X, \mathrm{b}, X), (X, \mathrm{c}, X), (X, \varepsilon, \varepsilon)\} \\
\text{(B)} &= \{(E, (\mathrm{a}\mid\mathrm{b}\mid\mathrm{c})^*\mathrm{aba}, X), (X, \mathrm{a}\mid\mathrm{b}\mid\mathrm{c}, X), (X, \varepsilon, \varepsilon)\} \\
\text{(D)} &= \{(E, (\mathrm{a}\mid\mathrm{b}\mid\mathrm{c})^*\mathrm{aba}(\mathrm{a}\mid\mathrm{b}\mid\mathrm{c})^*, \varepsilon)\}
\end{aligned}
$$

6 Reconhecimento de *tokens*

- Anteriormente vimos como se podem expressar padrões utilizando *expressões regulares*.
- Assim sendo, vamos definir as expressões regulares para os *tokens* do seguinte excerto duma linguagem: [2]

$$
\begin{aligned}
\textit{stmt} &\rightarrow \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt} \\
&\mid \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
&\mid \varepsilon \\
\textit{expr} &\rightarrow \textit{term}\ \textbf{relop}\ \textit{term} \\
&\mid \textit{term} \\
\textit{term} &\rightarrow \textbf{id} \\
&\mid \textbf{number}
\end{aligned}
$$

- Os símbolos terminais desta gramática são: **if**, **then**, **else**, **relop**, **id** e **number**.
- Os padrões para reconhecer estes *tokens* podem ser descritos com expressões regulares:

$$
\begin{aligned}
\textbf{if} &\rightarrow \text{if} \\
\textbf{then} &\rightarrow \text{then} \\
\textbf{else} &\rightarrow \text{else} \\
\textbf{relop} &\rightarrow \text{< | > | <= | >= | = | <>} \\
\textbf{digit} &\rightarrow \text{[0-9]} \\
\textbf{digits} &\rightarrow \textbf{digit}^+ \\
\textbf{number} &\rightarrow \textbf{digits}\ (\text{.}\ \textbf{digits})?\ (\text{E}\ [\text{+-}]?\ \textbf{digits})? \\
\textbf{letter} &\rightarrow \text{[A-Za-z]} \\
\textbf{id} &\rightarrow \textbf{letter}\ (\textbf{letter} \mid \textbf{digit})^*
\end{aligned}
$$

[2] Exemplo retirado do livro: *"Compilers: Principles, Techniques, & Tools", 2ed, Aho, et.al*

- Adicionalmente, o analisador léxico deve reconhecer e eliminar os caracteres correspondentes ao espaço em branco:

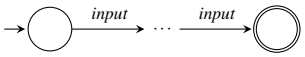
$$\textbf{ws} \quad \rightarrow \quad (\textbf{blank} \mid \textbf{tab} \mid \textbf{newline})^{+}$$

- Vamos tentar construir um analisador léxico que para além de reconhecer os *tokens* crie a seguinte informação:

tokens	Nome	Valor
ws	–	–
if	if	–
then	then	–
else	else	–
id	id	texto do identificador
number	number	texto do número
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

6.1 Diagramas de transição

- Como passo intermédio para a construção do analisador léxico, vamos converter "à mão" as expressões regulares em máquinas de estados (representadas por *diagramas de transição*)
- Veremos mais à frente que este processo pode ser sistematizado recorrendo a *autómatos finitos*.
- Os diagramas de transição contêm uma colecção de estados (representados por círculos).
- Cada estado representa uma sequência de condições que ocorreram no processo de reconhecimento léxico.
- Isto é, cada estado representa o que já aconteceu até esse ponto no reconhecimento da sequência de caracteres de entrada no analisador.
- As transições são dirigidas de um estado para outro, e são anotadas com a entrada que lhes corresponde.

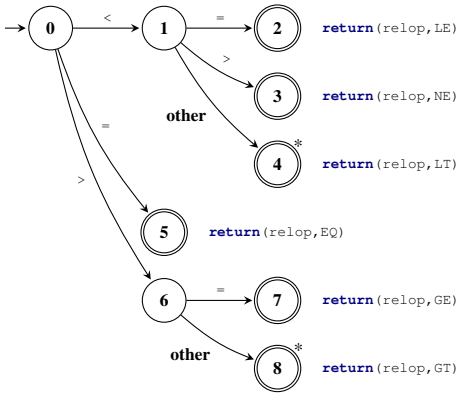


- As convenções a aplicar a estes diagramas são as seguintes:

  1. Cada estado é representado por um círculo e tem a si associado um rótulo que o identifica (em geral, um número ou uma letra).
  2. Alguns estados são considerados como *finais* (ou de aceitação). Estes estados indicam que um *token* foi reconhecido. Estes estados são representados com um círculo duplo.
  3. Se, por necessidade, tiver sido consumido um carácter a mais nesse processo de aceitação final, o nó que lhe corresponde será anotado com um asterisco.
  4. As transições entre estados são representadas por setas anotadas com o carácter que a despoleta.
  5. Um estado é designado como estado inicial, sendo indicado por uma transição (seta) sem estado de origem.

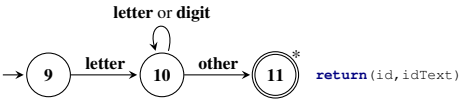
- Na construção destes diagramas vamos simplesmente enumerar novos estados por cada transição resultante de um carácter que, de alguma forma, avance no reconhecimento do *token*.

- O diagrama de transição para reconhecer os operadores relacionais pode ser o seguinte:



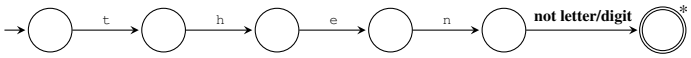
- Note que, para este tipo de *tokens*, este diagrama é uma estrutura de dados tipo árvore (deitada). Isso acontece em *tokens* fixos como o exemplificado ou as palavras reservadas.

- O diagrama de transição para identificadores:

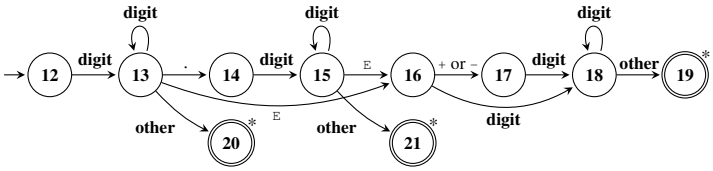


- O reconhecimento de identificadores pode levantar um problema de ambiguidade.
- De facto, as palavras reservadas da linguagem (ex: **then**) também podem ser reconhecidas como identificadores.
- Para resolver este problema, os analisadores léxicos dão prioridade a *tokens* que consomem mais caracteres e, em caso de conflito, estabelecem diferentes prioridades entre estes.
- Assim, o conflito entre identificadores e palavras reservadas é resolvido dando mais prioridade a estas.

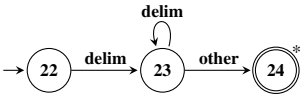
- O diagrama de transição para palavras reservadas é aqui exemplificado com o *token* **then**:



- O diagrama de transição para números:



- O diagrama de transição para espaço em branco:



- Agora podemos traduzir de uma forma quase automática os diagramas de transição para analisadores léxicos:

```
protected Token getRelop() {
   Token res = null;
   char c = 0;  boolean fail = false;  int state = 0;
   while(!fail && res == null) {
      switch(state) {
         case 0:
            c = nextChar();
            if (c == '<') state = 1;
            else if (c == '=') state = 5;
            else if (c == '>') state = 6;
            else { fail = true; retract(1); }
            break;
         case 1:
            ...
         case 2:
            res = new Token("relop", "LE");
            break;
        ...
```

```
            case 4:
                res = new Token("relop", "LT");
                retract(1);
                break;
            ...
        }
    }

    return res;
}
```

- Note que nos estados que requerem um carácter para fazerem uma transição $\{\mathbf{0},\mathbf{1},\mathbf{6}\}$, é invocada a função `nextChar`; assim como os estados com asterisco, o carácter a mais é reposto com a função `retract`
- Podemos implementar uma função por tipo de *token* (`getID`, `getReserved`, `getWS`, `getnumber`), e depois invocar sequencialmente cada uma delas (até que uma seja bem sucedida).

```
public Token nextToken() {
    Token res = getWS();
    if (res == null)
        res = getReserved();
    if (res == null)
        res = getID();
    if (res == null)
        res = getNumber();
    if (res == null)
        res = getRelop();
    if (EOF)
        res = new Token("EOF", "");
    else if (res == null)
        res = new Token("ERROR", "");
    return res;
}
```

- No entanto, esta solução não é a mais eficiente já quem na presença de falhas, estamos a rebobinar a fila de caracteres de entrada.
- Alternativamente, podemos tentar executar os vários diagramas em paralelo.
- Se se utilizar uma numeração diferente nos estados de cada *token* (como foi feito neste exemplo), a melhor solução será simplesmente juntar todas as máquinas de estados numa única.
- Esta solução não só pode ser automatizada, como também é bastante eficiente (isso pode ser medido pelo número de vezes em que é necessário voltar atrás no consumo de caracteres, i.e. nas invocações da função `retract`).
- As máquinas que permitem uma aproximação automática a este problema são os chamas *autómatos finitos* (como veremos, os diagramas de transição apresentados descrevem autómatos finitos deterministas incompletos).

```
protected Token get() {
    Token res = null;
    char c=0; String value=""; boolean fail=false; int state=0;
    while(!fail && res == null) {
        switch(state) {
            case 0:
                c = nextChar();
                if (c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else state = 9;
                break;
            ...
            case 2:
                res = new Token("relop", "LE");
                break;
            ...
            case 9:
                if (Character.isLetter(c)) {value+=c; state=10;}
                else state = 22;
                break;
```

```
            ...
            case 22:
                if (Character.isWhitespace(c)) state = 23;
                else {fail = true; retract(1);}
                break;
            ...
        }
    }
    return res;
}
```

7 Autómatos finitos

- Um autómato é uma "máquina" que executa sobre uma determinada sequência de entradas passo a passo (de forma discreta no tempo).
- Internamente, o autómato contém uma máquina de estados, que vai evoluindo até uma de duas possibilidades: aceitar ou rejeitar a entrada.
- A palavra de entrada será reconhecida se o estado final for de aceitação.
- Um autómato finito é caracterizado por definir uma máquina de estados finita, em que as transições de estados apenas têm em conta o estado actual e a entrada.
- Graficamente, associam-se círculos aos estados e anota-se as transições entre estados com as entradas correspondentes (estados de aceitação terão um círculo duplo).

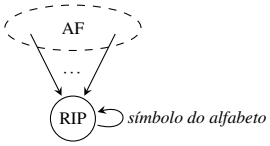
- Os autómatos finitos são classificados em dois tipos:

  a) *Autómato finito não determinista* (AFND): não existem restrições às condições colocadas nas transições. O mesmo símbolo (entrada) pode anotar várias transições a partir do mesmo estado, sendo também permitidas transições com a palavra vazia (ε).

  b) *Autómato finito determinista* (AFD): cada estado indica no máximo uma transição com cada símbolo do alfabeto.

- Qualquer um destes tipos de autómatos reconhece as mesmas linguagens, e demonstra-se que essas linguagens correspondem às linguagens regulares.[3]
- Note que um AFD é um caso particular de um AFND.
- Os autómatos finitos podem ser *completos* ou *incompletos*.

- Será *incompleto* caso não existam transições para todos os símbolos do alfabeto, e *completo* no caso contrário.
- Neste caso, o aparecimento desse símbolo com o autómato nesse estado, leva imediatamente à rejeição da palavra.
- Podemos sempre converter um autómato incompleto num completo destinando todas as transições não expressas para um novo estado que funcione como uma especie de "buraco negro". Isto é, um estado do qual não se pode sair. Obviamente, esse estado não pode ser de aceitação.



8 Autómato finito não determinista

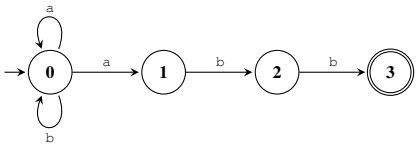
- Um autómato finito não determinista é um autómato finito onde:
  - as transições estão associadas a símbolos individuais do alfabeto ou à palavra vazia (ε);
  - de cada estado *saem zero ou mais transições* por cada símbolo do alfabeto ou ε;

[3] Com uma excepção menor: as linguagens regulares não expressam a linguagem vazia ($\emptyset$), o que pode ser trivialmente feito com autómatos finitos.

  - há um estado inicial;
  - há zero ou mais estados de aceitação, que determinam as palavras aceites;
  - uma dada palavra sobre o alfabeto faz o sistema avançar do estado inicial a *zero ou mais estados finais*, determinando estes a aceitação ou rejeição da palavra.
- Os arcos múltiplos permitem alternativas de reconhecimento.
- Os arcos ausentes representam quedas num estado de *morte* (estado não representado, logo implicando palavra não reconhecida).

AFND: exemplo

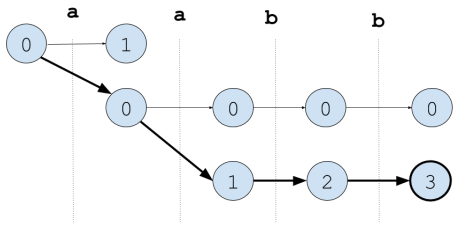
- Um possível diagrama de transição para um AFND que reconhece a expressão regular $(\mathbf{a}\,|\,\mathbf{b})^*\mathbf{abb}$ é o seguinte:



- É bem evidente o efeito do fecho de Kleene no diagrama, assim como a alternativa $(\mathbf{a}\,|\,\mathbf{b})$ e as sequências.

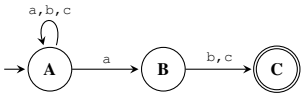
AFND: caminhos alternativos

- Será que a palavra **aabb** pertence esta linguagem?
- Existem 3 caminhos alternativos no diagrama:
  1. $0 \xrightarrow{\mathbf{a}} 1 \xrightarrow{\mathbf{a}} ?$
  2. $0 \xrightarrow{\mathbf{a}} 0 \xrightarrow{\mathbf{a}} 0 \xrightarrow{\mathbf{b}} 0 \xrightarrow{\mathbf{b}} 0$
  3. $0 \xrightarrow{\mathbf{a}} 0 \xrightarrow{\mathbf{a}} 1 \xrightarrow{\mathbf{b}} 2 \xrightarrow{\mathbf{b}} 3$
- Apenas o último termina no estado final.
- Podemos representar estes caminhos com uma estrutura tipo árvore (deitada):



AFND: exemplo

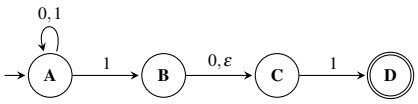
- Considerando o alfabeto $A = \{a, b, c\}$, que palavras são reconhecidas pelo autómato seguinte:



- Como expressão regular: $(\mathbf{a}\,|\,\mathbf{b}\,|\,\mathbf{c})^*\mathbf{a}\,(\mathbf{b}\,|\,\mathbf{c})$
- Como conjunto: $\mathbf{L} = \{\mathbf{w}\,\mathbf{a}\,\mathbf{X} : \mathbf{w} \in \mathbf{A}^* \wedge \mathbf{X} \in \{\mathbf{b}, \mathbf{c}\}\}$

AFND: exemplo com transições ε

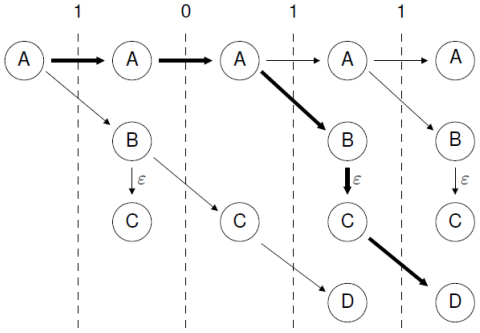
- Considere o seguinte AFND sobre o alfabeto $A = \{0, 1\}$:



- Será que a palavra **1011** é reconhecida?
- Há 6 caminhos possíveis:
  1. $A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{0}} A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{1}} A$
  2. $A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{0}} A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{1}} B$
  3. $A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{0}} A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{1}} B \xrightarrow{\varepsilon} C$
  4. $A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{0}} A \xrightarrow{\mathbf{1}} B \xrightarrow{\varepsilon} C \xrightarrow{\mathbf{1}} D$
  5. $A \xrightarrow{\mathbf{1}} B \xrightarrow{\mathbf{0}} C \xrightarrow{\mathbf{1}} D$
  6. $A \xrightarrow{\mathbf{1}} B \xrightarrow{\varepsilon} C$

AFND: exemplo com transições ε

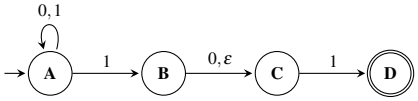
- Com a estrutura tipo árvore:



- A palavra é reconhecida uma vez que existe (pelo menos um) caminho que leva a **D**.

AFND: exemplo

- Que palavras são reconhecidas por este autómato?



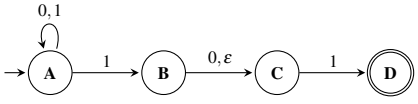
- Todas as palavras terminadas em **11** ou **101**.
- Como expressão regular: $(\mathbf{0}\,|\,\mathbf{1})^*\mathbf{10}?\mathbf{1}$

AFND: definição formal

- Um automato finito não determinista é um quíntuplo $M = (A, Q, q_0, \delta, F)$, em que[4]:
  - A é o alfabeto de entrada (sem a palavra vazia ε);
  - Q é um conjunto finito não vazio de estados;
  - $q_0 \in Q$ é o estado inicial;
  - $\delta \subseteq (Q \times A_\varepsilon \times Q)$ é a relação de transição entre estados, com $A_\varepsilon = A \cup \{\varepsilon\}$;
  - $F \subseteq Q$ é o conjunto dos estados de aceitação.

AFND: outro exemplo

- Represente analiticamente o AFND:



[4] δ é a letra grega minúscula delta.

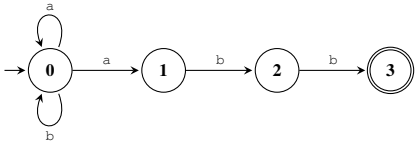
- O quíntuplo $M = (A, Q, q_0, \delta, F)$ é:
  - $A = \{\mathbf{0}, \mathbf{1}\}$
  - $Q = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$
  - $q_0 = \mathbf{A}$
  - $F = \{\mathbf{D}\}$
  - $\delta = \{(\mathbf{A}, \mathbf{0}, \mathbf{A}), (\mathbf{A}, \mathbf{1}, \mathbf{A}), (\mathbf{A}, \mathbf{1}, \mathbf{B}), (\mathbf{B}, \varepsilon, \mathbf{C}), (\mathbf{B}, \mathbf{0}, \mathbf{C}), (\mathbf{C}, \mathbf{1}, \mathbf{D})\}$
- Como expressão regular: $(\mathbf{0} \,|\, \mathbf{1})^* \mathbf{1} (\mathbf{0} \,|\, \varepsilon) \mathbf{1}$
- Ou alternativamente: $(\mathbf{0} \,|\, \mathbf{1})^* \mathbf{1} (\mathbf{0})? \mathbf{1}$

AFND: linguagem reconhecida

- Diz-se que um AFND $M = (A, Q, q_0, \delta, F)$, *aceita* uma palavra $u \in A^*$ se u se puder escrever na forma $u = u_1 u_2 \cdots u_n$, com $u_i \in A_\varepsilon$, e existir uma sequência de estados $s_0, s_1, \cdots, s_n$, que satisfaça as seguintes condições:
  1. $s_0 = q_0$
  2. qualquer que seja o $i = 1, \cdots, n$, $(s_{i-1}, u_i, s_i) \in \delta$
  3. $s_n \in F$
- Caso contrario diz-se que M *rejeita* a entrada.
- Note que n pode ser maior que $|u|$, porque alguns dos u_i podem ser ε.
- Usar-se-á a notação $q_i \xrightarrow{\mathbf{u}} q_j$ para representar a existência de uma palavra u que conduza do estado q_i ao estado q_j
- Usando esta notação tem-se $L(M) = \{u : q_0 \xrightarrow{\mathbf{u}} q_f \ \wedge \ q_f \in F\}$

8.1 Tabelas de transição

- Podemos representar um AFND por uma *tabela de transição*, em que as linhas correspondem aos estados, e as colunas aos símbolos de entrada incluindo a palavra vazia (A_ε).
- A tabela de transição para o AFND:

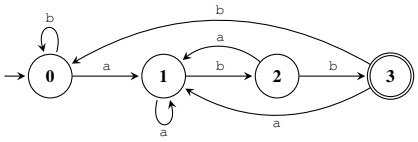


é a seguinte:

STATE	a	b	ε
$\rightarrow 0$	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3_f	$\emptyset$	$\emptyset$	$\emptyset$

9 Autómato finito determinista

- Um autómato finito determinista (AFD) é um caso especial de um AFND onde:
  - Não há transições com a palavra vazia (ε).
  - Para cada estado s e símbolo de entrada a existe no máximo uma transição de s anotada com a.
- Num AFD completo, existe uma transição para todos os símbolos do alfabeto.
- Um diagrama de transição para um AFD que reconhece a expressão regular $(\mathbf{a} \,|\, \mathbf{b})^* \mathbf{abb}$ é o seguinte:



Autómato finito determinista: tabela de transição

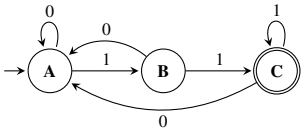
- Se estivermos a representar um AFD com uma tabela de transição, então cada entrada será um único estado (pelo que deixa de ser necessário representá-lo como conjunto):

STATE	a	b
$\rightarrow 0$	1	0
1	1	2
2	1	3
3_f	1	0

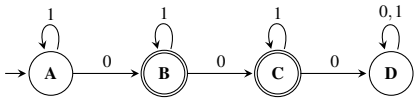
- Enquanto um AFND é uma representação abstracta dum algoritmo para reconhecer expressões regulares, o AFD é um algoritmo simples, concreto e muito eficiente para o mesmo fim.
- Felizmente é possível converter um AFND num AFD que reconhece a mesmo linguagem regular.

AFD: exemplo

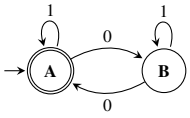
- Que palavras são reconhecidas pelo autómato seguinte:



- Todas as palavras terminadas em 11.
- Como expressão regular: $(\mathbf{0}\,|\,\mathbf{1})^*\mathbf{11}$
- Que palavras são reconhecidas pelo autómato seguinte:



- Todas as palavras com 1 ou 2 zeros.
- Como expressão regular: $\mathbf{1}^*\mathbf{01}^*\mathbf{0}?\mathbf{1}^*$
- Que palavras são reconhecidas pelo autómato seguinte:



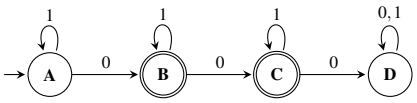
- Todas as palavras com um número par de zeros.
- Como expressão regular: $\mathbf{1}^*(\mathbf{01}^*\mathbf{0})^*\mathbf{1}^*$

AFD: definição formal

- Um automato finito determinista é um quíntuplo $M = (A, Q, q_0, \delta, F)$, em que:
  - A é o alfabeto de entrada (sem a palavra vazia ε);
  - Q é um conjunto finito não vazio de estados;
  - $q_0 \in Q$ é o estado inicial;
  - $\delta : Q \times A \rightarrow Q$ é uma função que determina a transição entre estados;
  - $F \subseteq Q$ é o conjunto dos estados de aceitação.
- Note que apenas muda a definição de δ relativamente aos AFND (agora é uma função).
- A função δ pode ser representada pelo conjunto de triplos $\in Q \times A \times Q$, ou pela tabela de transição (matriz de $|Q|$ linhas e $|A|$ colunas).

AFD: exemplo (2)

- Represente analiticamente o AFD:



- O quíntuplo $M = (A, Q, q_0, \delta, F)$ é:
  - $A = \{\mathbf{0}, \mathbf{1}\}$
  - $Q = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$
  - $q_0 = \mathbf{A}$
  - $F = \{\mathbf{B}, \mathbf{C}\}$

$\delta = \{(\mathbf{A},\mathbf{0},\mathbf{B}), (\mathbf{A},\mathbf{1},\mathbf{A}), (\mathbf{B},\mathbf{0},\mathbf{C}), (\mathbf{B},\mathbf{1},\mathbf{B}), (\mathbf{C},\mathbf{0},\mathbf{D}), (\mathbf{C},\mathbf{1},\mathbf{C}), (\mathbf{D},\mathbf{0},\mathbf{D}), (\mathbf{D},\mathbf{1},\mathbf{D})\}$

STATE	0	1
$\rightarrow A$	B	A
B_f	C	B
C_f	D	C
D	D	D

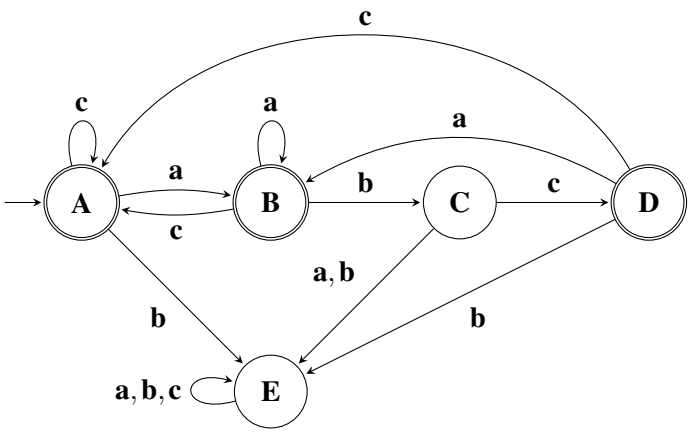
AFD: linguagem reconhecida

- Diz-se que um AFD $M = (A, Q, q_0, \delta, F)$, *aceita* uma palavra $u \in A^*$ se u se puder escrever na forma $u = u_1 u_2 \cdots u_n$ e existir uma sequência de estados $s_0, s_1, \cdots, s_n$, que satisfaça as seguintes condições:
  1. $s_0 = q_0$
  2. qualquer que seja o $i = 1, \cdots, n$, $s_i = \delta(s_{i-1}, u_i)$
  3. $s_n \in F$
- Caso contrario diz-se que M *rejeita* a entrada.

10 Autómato finito determinista

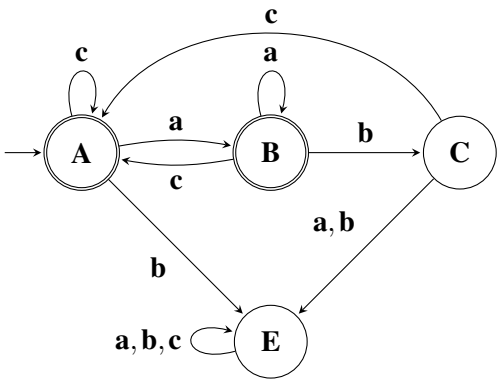
10.1 Projecto de autómato finito determinista

- Projete um AFD que reconheça as sequências definidas sobre o alfabeto $\mathbf{A} = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ que satisfazem o requisito de qualquer **b** ter um **a** imediatamente à sua esquerda e um **c** imediatamente à sua direita.
- Aproximação possível:
  - Note que para estar num estado final, caso apareça um símbolo **b** na entrada, é necessário garantir um estado prévio (**a**) e um estado seguinte (**c**).
  - Note também que esse estado seguinte é final (cumpre o requisito), o mesmo acontecendo com o estado inicial.
  - Temos assim a necessidade de pelo menos quatro estados (estado inicial, e os três estados correspondentes à sequência **abc**).
  - Por outro lado, caso apareça um símbolo **b**, sem que exista um **a** imediatamente à sua esquerda, ou um **c** imediatamente à sua direita, podemos desde logo afirmar que a entrada não cumpre o requirido pelo que precisamos de um estado que não seja final mas donde não seja possível sair (tipo "buraco negro").
- Chegamos assim ao seguinte AFD:



STATE	a	b	c
$\rightarrow A_f$	B	E	A
B_f	B	C	A
C	E	E	D
D_f	B	E	A
E	E	E	E

- Será que podemos simplificar esta autómato?
- Se compararmos os estados **A** e **D**, constata-se que são ambos finais e as transições para fora são equivalentes.
- logo podem ser fundidos:



STATE	a	b	c
$\rightarrow A_f$	B	E	A
B_f	B	C	A
C	E	E	A
E	E	E	E

10.2 Redução de autómatos finitos deterministas

Redução de AFD

- O exemplo anterior mostra que por vezes é possível simplificar os AFD reduzindo o número de estados.
- A ideia base é ter um procedimento sistemático para identificar estados equivalentes, fundindo-os num único estado.
- Dois estados são equivalentes se forem do mesmo tipo (final ou não final) e se todas as suas transições para o exterior forem iguais (i.e. para os mesmos estados).
- Formalmente podemos ir mais longe e afirmar que dois estados s_i e s_j de um autómato $M = (A, Q, q_0, \delta, F)$ são equivalentes se e só se

$$\forall_u \in A^* \quad \delta^*(s_i, u) \in F \Leftrightarrow \delta^*(s_j, u) \in F$$

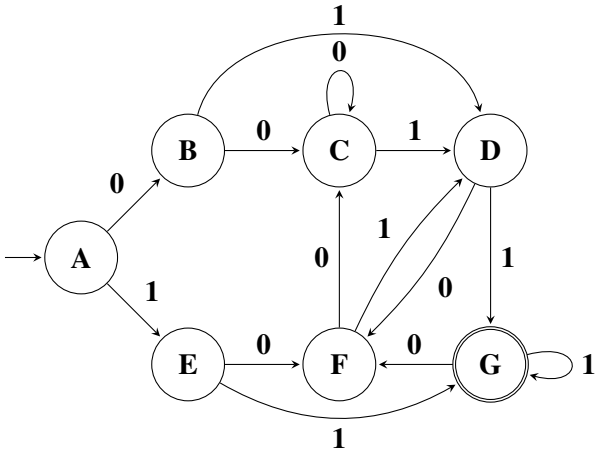
- Em que o fecho da função de transição $\delta^* : Q \times A \to Q$ é definido por:

$$\begin{aligned} \delta^*(q,\varepsilon) &= q \\ \delta^*(q,av) &= \delta^*(\delta(q,a),v), \quad \text{com} \quad a \in A \wedge v \in A^* \end{aligned}$$

- Note que esse autómato M aceita uma sequência de símbolos u se $\delta^*(q_0,u) \in F$.
- A linguagem reconhecida por M – $L(M)$ – é definida por

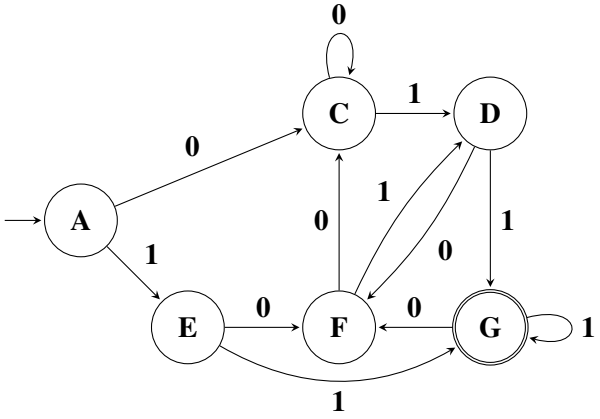
$$L(M) = \{u \in A^* \mid M \text{aceita} u\} = \{u \in A^* \mid \delta^*(q_0,u) \in F\}$$

- Considere o seguinte autómato:



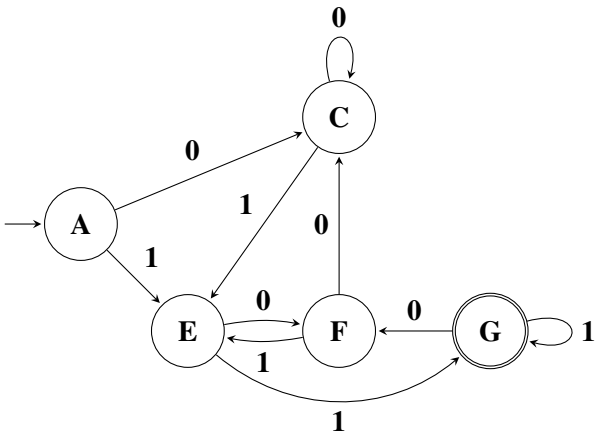
STATE	**0**	**1**
$\to A$	B	E
B	C	D
C	C	D
D	F	G
E	F	G
F	C	D
G_f	F	G

- Vamos primeiro aplicar a regra de fundir estados na mesma situação (mesmo tipo e transições iguais).
- Olhando para a tabela de transições constata-se que os estados **B** e **C** são equivalentes:



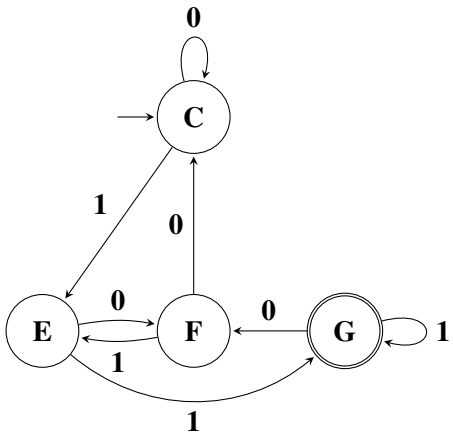
STATE	**0**	**1**
$\to A$	C	E
C	C	D
D	F	G
E	F	G
F	C	D
G_f	F	G

- Temos também equivalência nos estados **D** e **E**:



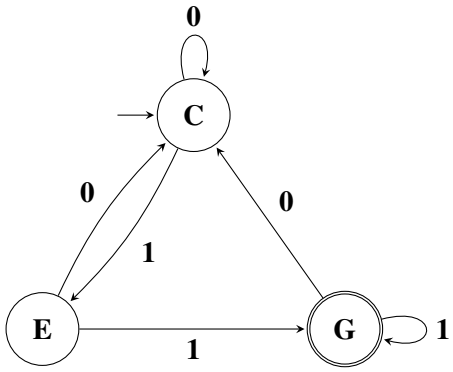
STATE	**0**	**1**
$\to A$	C	E
C	C	E
E	F	G
F	C	E
G_f	F	G

- Agora há equivalência nos estados **A** e **C**:



STATE	**0**	**1**
$\rightarrow C$	C	E
E	F	G
F	C	E
G_f	F	G

- Por fim há equivalência nos estados **C** e **F**:



STATE	**0**	**1**
$\rightarrow C$	C	E
E	C	G
G_f	C	G

- Note que os estados **E** e **G** embora tendo as mesmas transições, não são equivalentes.
- No entanto, este método para simplificar AFDs não garante um minimização total, já que não lida com o problema de poder haver ciclos entre estados equivalentes.
- Esse problema é resolvido com o algoritmo que se apresenta a seguir.

Algoritmo de redução de AFD

- Procedimento:

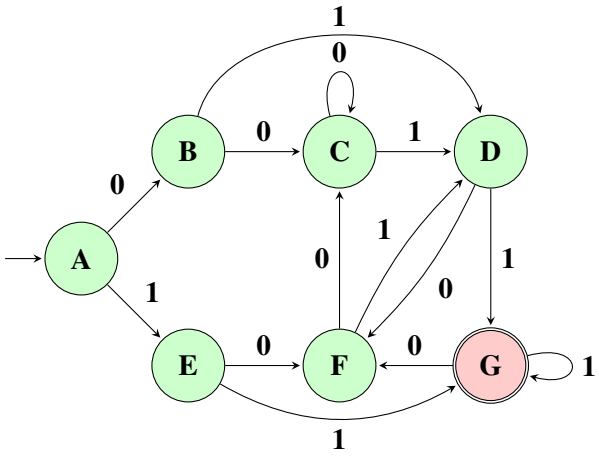
  1. Primeiro divide-se os estados em dois conjuntos: o conjunto dos estados finais (aceitação) e o conjunto com os restantes estados;[5]
  2. Depois vai-se particionando sucessivamente os conjuntos existentes, sempre que dentro do conjunto existam estados que tenham transições com o mesmo símbolo para diferentes conjuntos.
  3. O passo anterior é repetido até que não sejam possíveis mais partições. Nessa situação, o AFD está minimizado.

- Recuperando o exemplo anterior, temos como conjuntos de partida:

$$\begin{aligned} C_1 &= Q - F = \{A,B,C,D,E,F\} \\ C_2 &= F = \{G\} \end{aligned}$$

[5] Se o AFD for incompleto, então primeiro tem de ser transformado num completo.

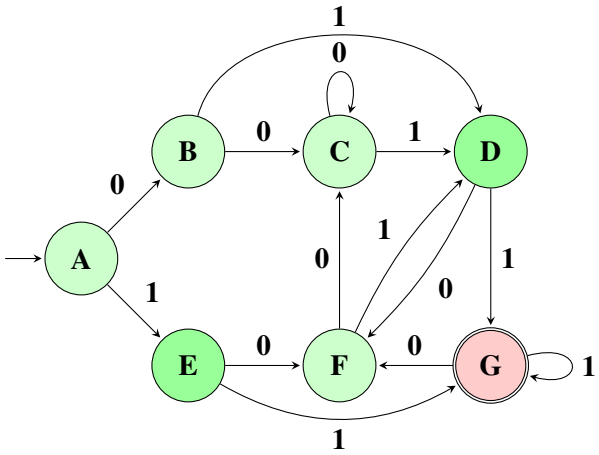
C_1 C_2



SET/STATE	0	1
$\rightarrow C_1/A$	C_1	C_1
C_1/B	C_1	C_1
C_1/C	C_1	C_1
C_1/D	C_1	C_2
C_1/E	C_1	C_2
C_1/F	C_1	C_1
C_2/G_f	C_1	C_2

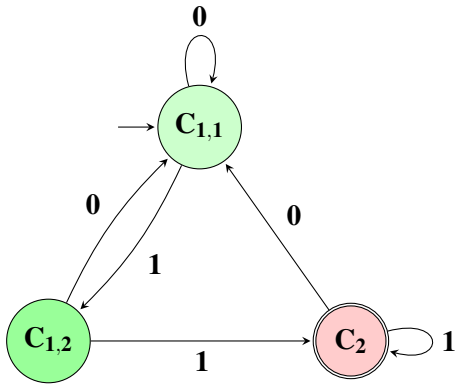
- O conjunto C_1 tem de ser partido em dois, já que para a entrada **1** os estados **D** e **E** têm uma transição para um conjunto diferente (C_2) do que os restantes estados.

$C_{1,1}$ $C_{1,2}$ C_2



SET/STATE	0	1
$\rightarrow C_{1,1}/A$	$C_{1,1}$	$C_{1,2}$
$C_{1,1}/B$	$C_{1,1}$	$C_{1,2}$
$C_{1,1}/C$	$C_{1,1}$	$C_{1,2}$
$C_{1,1}/F$	$C_{1,1}$	$C_{1,2}$
$C_{1,2}/D$	$C_{1,1}$	C_2
$C_{1,2}/E$	$C_{1,1}$	C_2
C_2/G_f	$C_{1,1}$	C_2

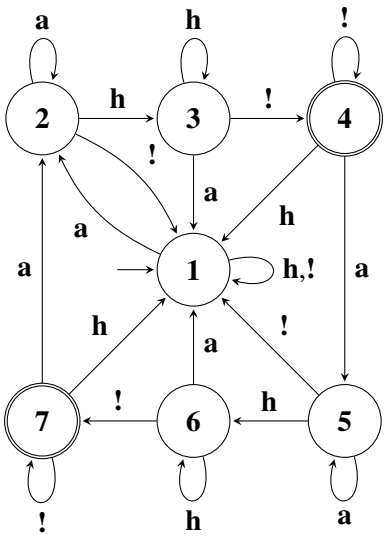
- Chegamos assim a um AFD minimizado equivalente ao que já tínhamos chegado:



STATE	0	1
$\rightarrow C_{1,1}$	$C_{1,1}$	$C_{1,2}$
$C_{1,2}$	$C_{1,1}$	C_2
$C_{2,f}$	$C_{1,1}$	C_2

Redução de AFD: exemplo 2

- Considere o seguinte autómato para reconhecer frases tipo **ah! ah!**:



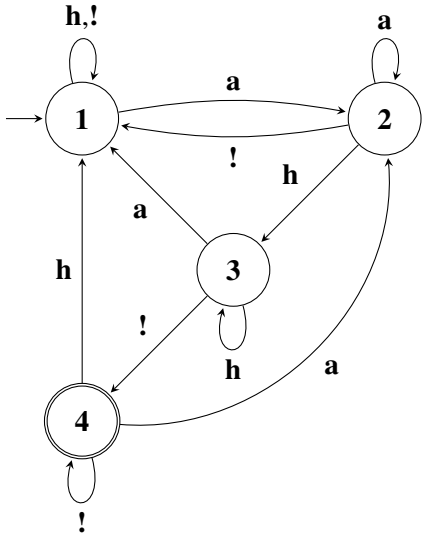
STATE	a	h	!
$\rightarrow 1$	2	1	1
2	2	3	1
3	1	3	4
4_f	5	1	4
5	5	6	1
6	1	6	7
7_f	2	1	7

STATE		a	h	!
$\rightarrow 1$	C_1	C_1	C_1	C_1
2	C_1	C_1	C_1	C_1
3	C_1	C_1	C_1	C_2
5	C_1	C_1	C_1	C_1
6	C_1	C_1	C_1	C_2
4_f	C_2	C_1	C_1	C_2
7_f	C_2	C_1	C_1	C_2

STATE		a	h	!
$\rightarrow 1$	$C_{1,1}$	$C_{1,1}$	$C_{1,1}$	$C_{1,1}$
2	$C_{1,1}$	$C_{1,1}$	$C_{1,2}$	$C_{1,1}$
5	$C_{1,1}$	$C_{1,1}$	$C_{1,2}$	$C_{1,1}$
3	$C_{1,2}$	$C_{1,1}$	$C_{1,2}$	C_2
6	$C_{1,2}$	$C_{1,1}$	$C_{1,2}$	C_2
4_f	C_2	$C_{1,1}$	$C_{1,1}$	C_2
7_f	C_2	$C_{1,1}$	$C_{1,1}$	C_2

STATE		a	h	!
$\rightarrow 1$	$C_{1,1,1}$	$C_{1,1,2}$	$C_{1,1,1}$	$C_{1,1,1}$
2	$C_{1,1,2}$	$C_{1,1,2}$	$C_{1,2}$	$C_{1,1,1}$
5	$C_{1,1,2}$	$C_{1,1,2}$	$C_{1,2}$	$C_{1,1,1}$
3	$C_{1,2}$	$C_{1,1,1}$	$C_{1,2}$	C_2
6	$C_{1,2}$	$C_{1,1,1}$	$C_{1,2}$	C_2
4_f	C_2	$C_{1,1,2}$	$C_{1,1,1}$	C_2
7_f	C_2	$C_{1,1,2}$	$C_{1,1,1}$	C_2

- Donde resulta o seguinte autómato final:



STATE	a	h	!
$\rightarrow 1$	2	1	1
2	2	3	1
3	1	3	4
4_f	2	1	4

11 Conversão de AFND em AFD

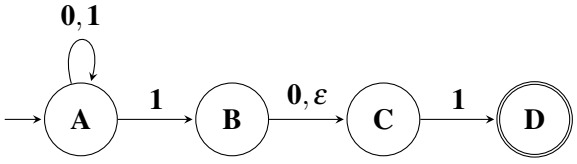
- Como já foi referido um AFD é um AFND, mas o contrário não é necessariamente verdadeiro.
- Nos AFD as transições são funções e as tabelas de transição são mais simples havendo sempre uma transição para no máximo um único estado.
- Assim, em geral, a implementação de AFD é preferível à implementação de AFND.
- É sempre possível converter um AFND num AFD.
- Vamos ver um algoritmo genérico para esse efeito.
- A ideia geral do algoritmo resulta da constatação de que num AFND as transições fazem-se de um subconjunto dos seus estados para outro subconjunto.
- Assim podemos transformar um AFND num AFD tomando como estados os subconjuntos do AFND.
- Com esta estratégia, para um AFND com n estados no pior caso podemos ter 2^n estados no AFD.
- No entanto, em geral constata-se que o número de estados é da mesma ordem de grandeza.

- O algoritmo assenta na construção, passo a passo, da tabela de transição para o AFD, partindo do AFND.
- Considerando que: **s** é um qualquer estado do AFND, **C** é um conjunto de estados do AFND e **a** é um qualquer símbolo de entrada, então:

Operação	Descrição
ε-closure($\mathbf{s}$)	Conjunto de estados do AFND para os quais pode haver uma transição a partir do estado **s** apenas pela palavra vazia (ε).
ε-closure($\mathbf{C}$)	Conjunto de estados do AFND para os quais pode haver uma transição a partir de qualquer estado do conjunto **C** apenas pela palavra vazia.
move($\mathbf{C},\mathbf{a}$)	Conjunto de estados do AFND para os quais pode haver uma transição a partir de qualquer estado do conjunto **C** pelo símbolo de entrada **a**.

- O estado inicial do AFD será o resultante da aplicação de ε-closure ao estado inicial do AFND.
- Os estados finais do AFD, serão todos os estados que contiverem pelo menos um estado final do AFDN.

- Como exemplo, vamos considerar o AFND seguinte:



STATE	**0**	**1**	ε
$\rightarrow A$	$\{A\}$	$\{A,B\}$	$\emptyset$
B	$\{C\}$	$\emptyset$	$\{C\}$
C	$\emptyset$	$\{D\}$	$\emptyset$
D_f	$\emptyset$	$\emptyset$	$\emptyset$

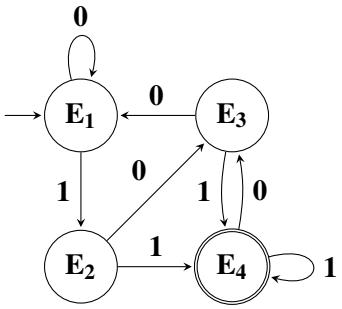
- Vamos identificar os estados do AFD por $E_i,\ i \in \mathbb{N}$
- O estado inicial para o AFD será dado por: $\varepsilon\text{-closure}(\{\mathbf{A}\}) = \{\mathbf{A}\} = E_1$

- Seguidamente vamos determinar os subconjuntos de estados AFND que resultam da transição de E_1 por cada símbolo do alfabeto.
  $\varepsilon\text{-closure}(\text{move}(E_1,\mathbf{0})) = \varepsilon\text{-closure}(\{\mathbf{A}\}) = \{\mathbf{A}\} = E_1$
- Como chegámos a um subconjunto que já existe ($\{\mathbf{A}\}$), não há lugar à criação de um novo estado para o AFD.
  $\varepsilon\text{-closure}(\text{move}(E_1,\mathbf{1})) = \varepsilon\text{-closure}(\{\mathbf{A},\mathbf{B}\}) = \{\mathbf{A},\mathbf{B},\mathbf{C}\} = E_2$
- Agora temos um novo subconjunto pelo que é necessário um novo estado (E_2).
- Aplicamos agora a mesma receita a esse novo estado até que não resultem novos estados (situação em que teremos o AFD equivalente ao AFND de que partimos).
  $\varepsilon\text{-closure}(\text{move}(E_2,\mathbf{0})) = \varepsilon\text{-closure}(\{\mathbf{A},\mathbf{C}\}) = \{\mathbf{A},\mathbf{C}\} = E_3$

ε-closure(move(E_2, **1**)) = ε-closure({**A**, **B**, **D**}) = {**A**, **B**, **C**, **D**} = E_4

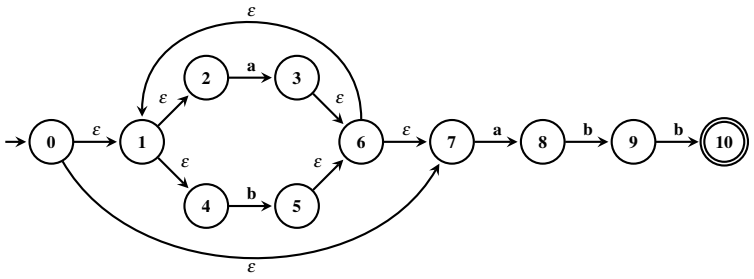
ε-closure(move(E_3, **0**)) = ε-closure({**A**}) = {**A**} = E_1
ε-closure(move(E_3, **1**)) = ε-closure({**A**, **B**, **D**}) = {**A**, **B**, **C**, **D**} = E_4
ε-closure(move(E_4, **0**)) = ε-closure({**A**, **C**}) = {**A**, **C**} = E_3
ε-closure(move(E_4, **1**)) = ε-closure({**A**, **B**, **D**}) = {**A**, **B**, **C**, **D**} = E_4

- Logo, vamos ter o seguinte AFD:



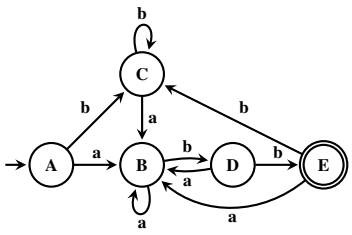
AFND	AFD	**0**	**1**
{**A**}	$\rightarrow E_1$	E_1	E_2
{**A**, **B**, **C**}	E_2	E_3	E_4
{**A**, **C**}	E_3	E_1	E_4
{**A**, **B**, **C**, **D**}	$E_{4,f}$	E_3	E_4

- A AFND seguinte foi uma implementação (que, como veremos, resulta directamente da aplicação de um algoritmo) da expressão regular: $(\mathbf{a}\,|\,\mathbf{b})^*\mathbf{abb}$



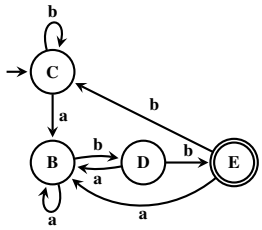
STATE	**a**	**b**	ε
$\rightarrow 0$	$\emptyset$	$\emptyset$	{1, 7}
1	$\emptyset$	$\emptyset$	{2, 4}
2	{3}	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	{6}
4	$\emptyset$	{5}	$\emptyset$
5	$\emptyset$	$\emptyset$	{6}
6	$\emptyset$	$\emptyset$	{1, 7}
7	{8}	$\emptyset$	$\emptyset$
8	$\emptyset$	{9}	$\emptyset$
9	$\emptyset$	{10}	$\emptyset$
10_f	$\emptyset$	$\emptyset$	$\emptyset$

- Estado inicial: ε-closure({**0**}) = {**0**, **1**, **2**, **4**, **7**} = A
ε-closure(move(A, **a**)) = ε-closure({**3**, **8**}) = {**1**, **2**, **3**, **4**, **6**, **7**, **8**} = B
ε-closure(move(A, **b**)) = ε-closure({**5**}) = {**1**, **2**, **4**, **5**, **6**, **7**} = C
ε-closure(move(B, **a**)) = ε-closure({**3**, **8**}) = {**1**, **2**, **3**, **4**, **6**, **7**, **8**} = B
ε-closure(move(B, **b**)) = ε-closure({**5**, **9**}) = {**1**, **2**, **4**, **5**, **6**, **7**, **9**} = D
ε-closure(move(C, **a**)) = ε-closure({**3**, **8**}) = {**1**, **2**, **3**, **4**, **6**, **7**, **8**} = B
ε-closure(move(C, **b**)) = ε-closure({**5**}) = {**1**, **2**, **4**, **5**, **6**, **7**} = C
ε-closure(move(D, **a**)) = ε-closure({**3**, **8**}) = {**1**, **2**, **3**, **4**, **6**, **7**, **8**} = B
ε-closure(move(D, **b**)) = ε-closure({**5**, **10**}) = {**1**, **2**, **4**, **5**, **6**, **7**, **10**} = E
ε-closure(move(E, **a**)) = ε-closure({**3**, **8**}) = {**1**, **2**, **3**, **4**, **6**, **7**, **8**} = B
ε-closure(move(E, **b**)) = ε-closure({**5**}) = {**1**, **2**, **4**, **5**, **6**, **7**} = C



AFND	AFD	**a**	**b**
{**0**, **1**, **2**, **4**, **7**}	$\rightarrow A$	B	C
{**1**, **2**, **3**, **4**, **6**, **7**, **8**}	B	B	D
{**1**, **2**, **4**, **5**, **6**, **7**}	C	B	C
{**1**, **2**, **4**, **5**, **6**, **7**, **9**}	D	B	E
{**1**, **2**, **4**, **5**, **6**, **7**, **10**}	E_f	B	C

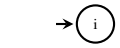
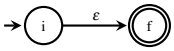
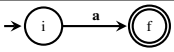
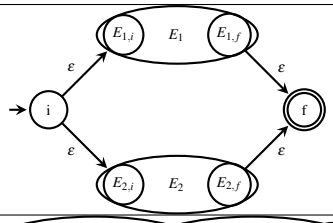
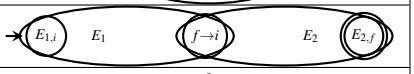
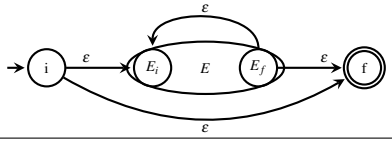
- Este AFD pode ainda ser minimizado (os estados **A** e **C** são equivalentes):



AFD	a	b
B	B	D
$\rightarrow C$	B	C
D	B	E
E_f	B	C

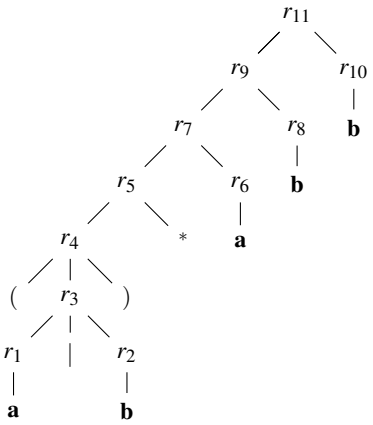
12 Conversão de uma expressão regular num AFND

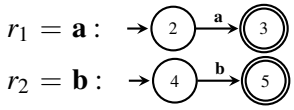
- Para compreendermos minimamente a construção de analisadores léxicos só falta abordarmos o problema da conversão automática de expressões regulares para autómatos.
- Para esse fim vamos apresentar um algoritmo (*McNaughton-Yamada-Thompson*) que converte uma qualquer `ER` num `AFND`.
- A estratégia baseia-se no seguinte:
  - Ter AFND definidos para `ER` elementares;
  - Ter padrões para AFND resultantes das operações sobre `ER` (reunião, concatenação, fecho de Kleene, . . . , . . .).
  - Construir o AFND recorrendo à árvore sintáctica da `ER`.
- A tabela seguinte mostra os padrões para AFND de `ER`

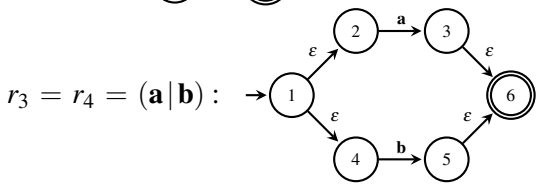
Descrição	ER	AFND
Linguagem vazia	()	
Palavra vazia	ε	
Símbolo do alfabeto	**a**	
União de AFND	$(E_1 \mid E_2)$	
Concatenação de AFND	E_1E_2	
Fecho de Kleene de AFND	E^*	

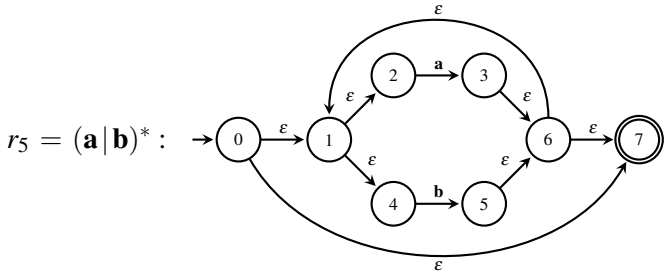
Conversão de uma ER num AFND: Exemplo

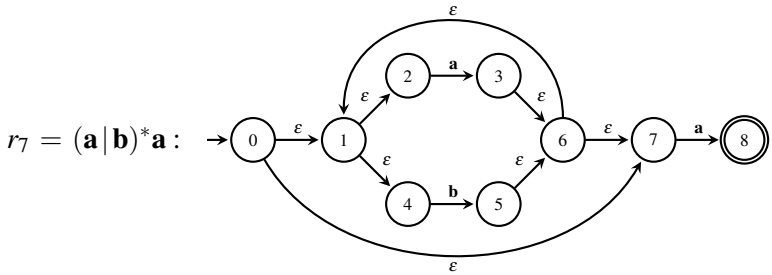
- Vamos então construir um AFND para a `ER`: $(\mathbf{a} \mid \mathbf{b})^*\mathbf{abb}$
- A árvore sintáctica desta `ER` é a seguinte:

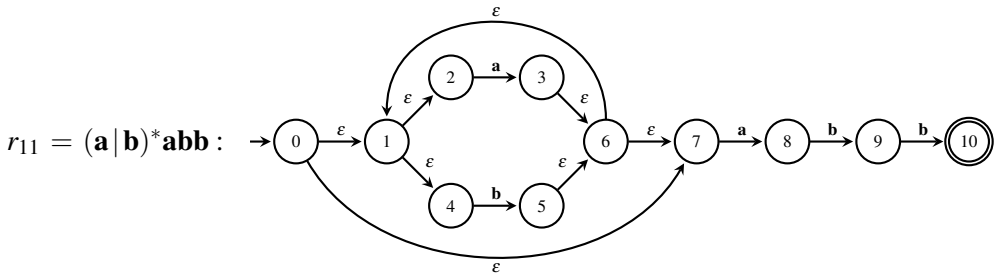








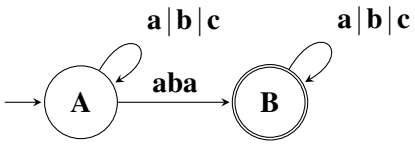




13 Autómato finito generalizado (AFG)

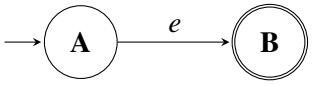
- Nos autómatos finitos apresentados as transições entre estados apenas decorrem de símbolos do alfabeto ou, no caso dos AFND, da palavra vazia (ε).
- No entanto podemos aproximar ainda mais os autómatos finitos das expressões regulares fazendo

com que as transições possam decorrer de ER (nas quais os símbolos do alfabeto e a palavra vazia são casos elementares).
- Este tipo de autómatos designa-se por *Autómato finito generalizado* (AFG).
- Por exemplo, um AFG sobre o alfabeto $A = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ para o conjunto de palavras que contém a palavra **aba** será:



13.1 AFG reduzido

- Um AFG com a forma



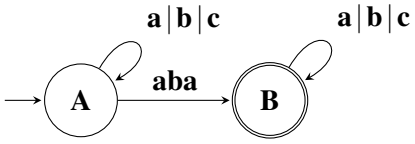
designa-se por *autómato finito generalizado reduzido*.
- Note que:
  - O estado *A* não é de aceitação e não tem arcos a chegar de outros estados.
  - O estado *B* é de aceitação e não tem arcos a sair.
- Se reduzir um AFG à forma anterior a expressão – *e* – é uma expressão regular equivalente ao autómato.

13.2 Conversão de uma AFG numa ER

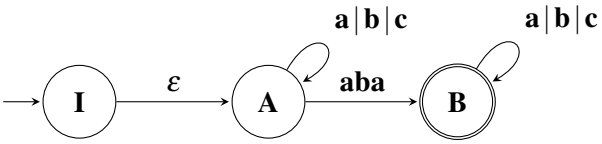
- Assim transformar uma AFG num AFG reduzido corresponde a determinar a ER que lhe é equivalente.
- Algoritmo de conversão:
  1. Transformação de um AFG noutro cujo estado inicial não tenha arcos a chegar.
     - Se necessário, acrescenta-se um novo estado inicial com um arco em ε para o antigo.
  2. Transformação de um AFG noutro com um único estado de aceitação, sem arcos de saída.
     - Se necessário, acrescenta-se um novo estado, que passa a ser o único de aceitação, que recebe arcos em ε dos anteriores estados de aceitação, que deixam de o ser.
  3. Eliminação dos restantes estados.
     - Os estados são eliminados um a um, em processos de transformação que mantêm a equivalência.

Conversão de uma AFG numa ER: Exemplo

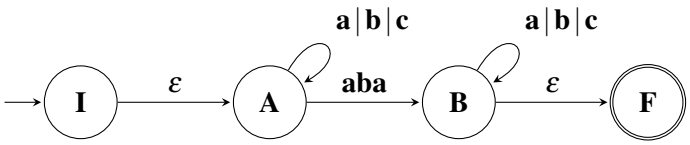
- Recuperando o AFG atrás apresentado vamos aplicar este algoritmo para o transformar numa ER.



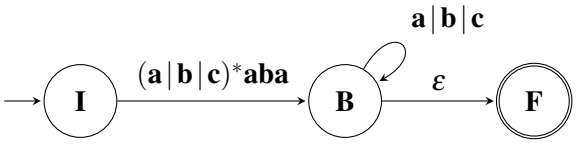
- Aplicando a regra 1:



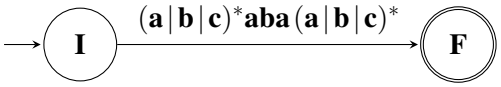
- Aplicando a regra 2:



- Eliminando o estado *A* aplicando a regra 3:



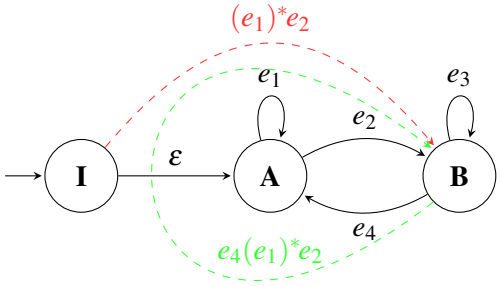
- Por fim, eliminando o estado *B* aplicando novamente a regra 3:



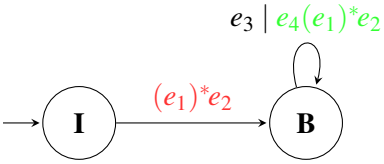
Eliminar estado com arcos a chegar de outros estados

- Se for necessário eliminar um estado que seja destino de arcos de outros estados, é necessário garantir – nesses estados – que o caminho de reconhecimento garantido pelo estado a eliminar não se altera.

4. Considerando que (e_1, e_2, e_3, e_4) são ER, a eliminação de estado A do AFG

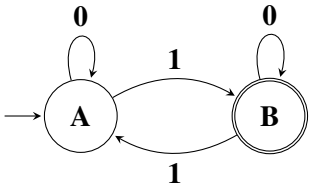


resulta no seguinte AFG:

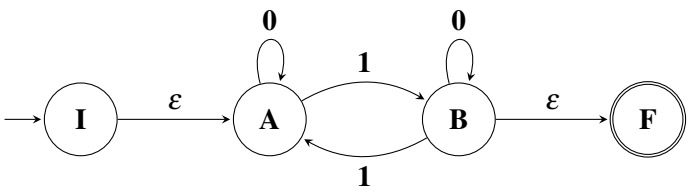


Conversão de uma AFG numa ER: Exemplo 2

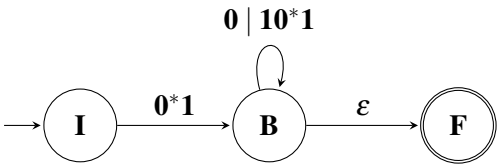
- Obtenha uma ER equivalente ao AF seguinte:



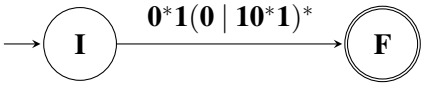
- Regras 1 e 2:



- Eliminar o estado A pela regra 4:



- Finalmente, eliminar o estado B pela regra 3:



- Logo a ER equivalente será: $\mathbf{0}^*\mathbf{1}(\mathbf{0} \mid \mathbf{10}^*\mathbf{1})^*$