Tema 01

Compiladores, Linguagens e Gramáticas

Enquadramento e Introdução

Linguagens Formais e Autómatos, 2º semestre 2017-2018

Miguel Oliveira e Silva, DETI, Universidade de Aveiro

Conteúdo

1 Enquadramento

- 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 a 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.
- Por outro lado, a nossa compreensão de um texto não se faz compreendendo 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 composto por mais do que uma linguagem:
  - Uma que explicita as regras para construir palavras a partir do alfabeto das letras:

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

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

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

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

- Inerente às linguagens, é a necessidade de decidir se uma sequência de símbolos do alfabeto é válida.
- 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!

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

2 Compiladores: Introdução

- 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*

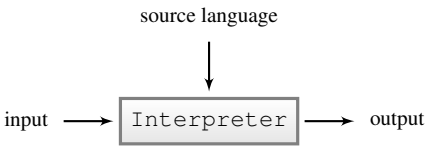
```
public class Hello
{
  public static void main(String[] args)
  {
    System.out.println("Hello!");
  }
}
```

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

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2
       3: ldc           #3
       5: invokevirtual #4
       8: return
}
```

- Uma variante possível consiste num *interpretador*:

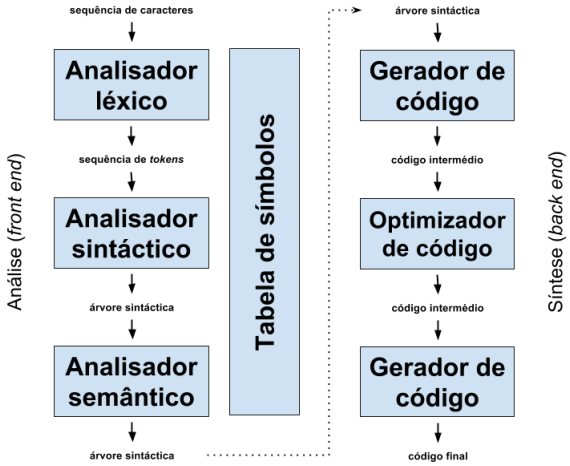


- Neste caso a execução é feita instrução a instrução.

- Existem também aproximações hibridas em 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, ...).

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 são eliminados pelo analisador lexical.
- Exemplo 2: esboço de linguagem de processamento geométrico:

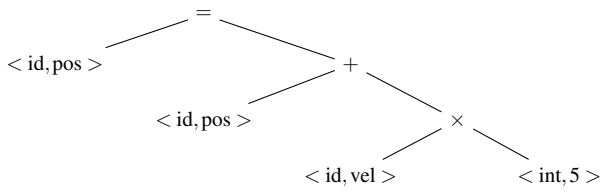
```
distance (0,0) (4,3)
```

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

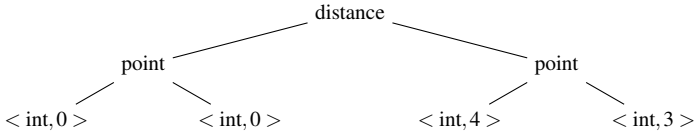
```
<distance> <(> <num,0> <,> <num,0> <)>
<(> <num,4> <,> <num,3> <)>
```

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.
- Qualquer que seja a linguagem que se pretende 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:



- No exemplo 2:



- 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 todas restantes 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.
- Se no exemplo 2 existisse a instrução de um círculo do qual fizesse parte a definição do seu raio, então poderíamos ter como regra semântica este valor não poder ser negativo.
- 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).

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)
MULT  R2, R2, #5.0
LOAD  R1, id(pos)
ADD   R1, R1, R2
STORE id(pos), 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
```

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>
```

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;

}
```

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+"\"");
                            exit(1);
                        }
                    }
                    break;
...
    }
...
```

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;
}
```

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;
}
```

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).

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;
}
```

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 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

Linguagens

- 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* 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*.
- Seja como for, podemos 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 exaustão é o conjunto dos algarismos binários $\{0, 1\}$.
- Na definição por compreensão utiliza-se a seguinte notação:

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

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;
    }
...
```

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.
    $2016, 234523, 999999999999999, 0$
  - $A = \{0,1,\cdots,0,a,b,\cdots,z,@,\cdots\}$
    $\text{mos@ua.pt}, \text{Bom dia!}$

- 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 $A = \{0,1\}$
  - $L_1 \;=\; \{u \mid u \in A^* \wedge |u| \leq 2\} \;=\; \{\varepsilon,0,1,00,01,10,11\}$
  - $L_2 \;=\; \{u \mid u \in A^* \wedge \forall_i u_i = 0\} \;=\; \{0,00,000,0000,\cdots\}$

- $L_3 = \{u \mid u \in A^* \wedge u.\text{count}(1)\ mod\ 2 = 0\} = \{000, 11, 000110101, \cdots\}$
- $L_4 = \{\} = \emptyset$
- $L_5 = \{\varepsilon\}$
- $L_6 = A$
- $L_7 = A^*$

- 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)
  - $|u| > 0 \wedge |v| > 0 \implies u.v \neq v.u$ (não comutatividade)
- 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 $A = \{a, b\}$:

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

- qual será o resultado da 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 $A = \{a, b\}$:

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

- qual será o resultado da 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), e contém mais informação (semântica) do que a frase original.
- 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.
- 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.

Gramáticas: exemplos

- A gramática construída sobre a frase "O cão ladra" será:

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

- P é constituído pelas regras já apresentadas:

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

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

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

- 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:

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

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

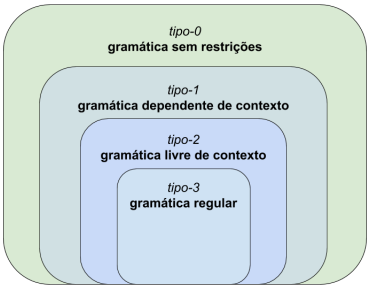
$$L_3 = \{u \mid u \in A^* \wedge u.\text{count}(1)\ mod\ 2 = 0\}$$

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

$$\begin{aligned} S &\rightarrow S\,1\,S\,1\,S \mid A \\ A &\rightarrow 0\,A \mid \varepsilon \end{aligned}$$

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 $|\alpha| \leq |\beta|$.
  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.



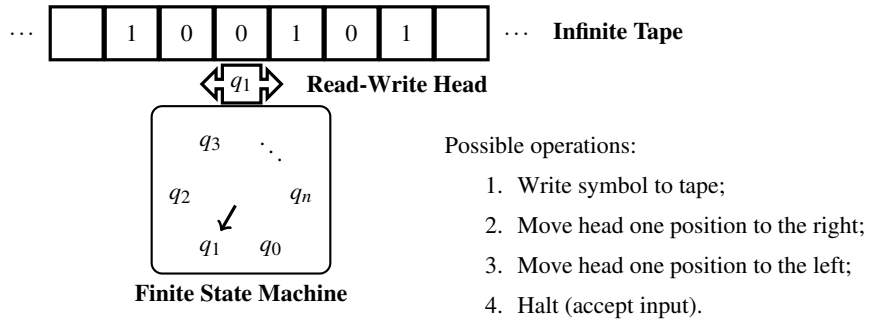
- Para cada um desses tipos podem ser definidos diferentes tipos de máquinas (algoritmos) que as podem reconhecer.
- Quanto mais simples for a gramática, mas simples e eficiente é a máquina que reconhece essas linguagens.

- 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.
- 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 (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).
- Por exemplo: $3+2$ a que corresponde o seguinte estado na fita (símbolo a negrito é a posição da "cabeça"): $\cdots\mathbf{0}111101110\cdots$
- 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;

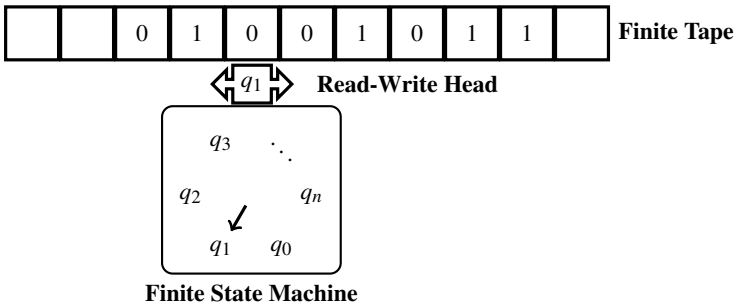
  1 escrever o símbolo 1 na fita;

  h aceitar e terminar autómato.

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

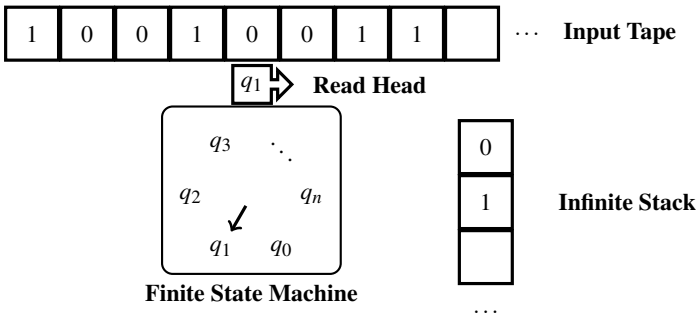
	Entrada	
Estado	**0**	**1**
E_1	E_1/d	E_2/d
E_2	$E_3/1$	E_2/d
E_3	$--$	E_4/d
E_4	E_5/e	E_4/d
E_5	$--$	$E_6/0$
E_6	E_7/e	$--$
E_7	$--$	$E_8/0$
E_8	E_8/h	$--$

6.2.2 Autómatos linearmente limitados



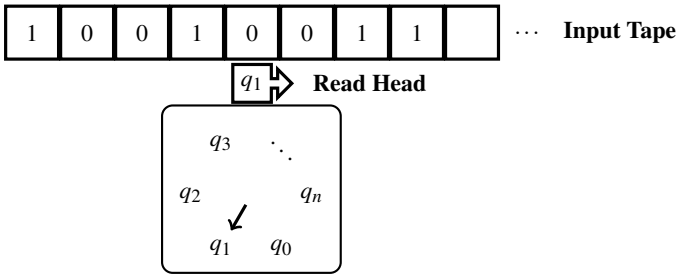
- Diferem das MT pela finitude da fita.

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.