Expressões regulares:

Padrão	Símbolo
Instancia no mínimo *n* e no máximo *m* ocorrências de *x*	`x{n,m}`
Instancia exatamente *n* ocorrências de *x*	`x{n}`
Instancia pelo menos *n* ocorrências de *x*	`x{n,}`
A instanciação deve ocorrer no inicio da informação	`^pattern`
A instanciação deve ocorrer no final da informação	`pattern$`
Um carater do conjunto	`[carateres]`
Um qualquer carater exceto os do conjunto	`[^carateres]`
Intervalo	`[char1-char2]`
Agrupamento	`(sub-pattern)`
Ou	`|`
Carater de escape para carateres com significado especial	`\`

Padrão	Símbolo	Alternativa
Um digito	`\d`	`[0123456789] ou [0-9]`
Um qualquer carater que não seja digito	`\D`	`[^0-9]`
Um carater alfanumérico	`\w`	`[a-zA-Z0-9_]`
Um qualquer carater que não seja alfanumérico	`\W`	`[^\w]`
Um carater branco	`\s`	`[ \t\n\r\f]`
Um qualquer carater que não seja branco	`\S`	`[^\s]`
Um qualquer carater exceto a mudança de linha	`.`	`[^\n]`
Instancia com uma ou mais ocorrências de *x*	`x+`	`{1,}`
x pode existir ou não	`x?`	
Instancia com zero ou mais ocorrências de *x*	`x*`	`{0,}`

Exemplos. **Descreva informalmente as linguagens representadas pelas seguintes expressões:**

0(0|1)*0 - Conj. de palavras sobre o alfabeto ∑={0,1} que começam e terminam com 0

(01)*- Conj. de palavras sobre o alfabeto∑={0,1}composto por tds as repetições da sequência 01 e a palavra vazia-ε

0*10*10*10*-Conjunto de palavras sobre o alfabeto ∑={0,1} com três 1's

Exemplos: **Considerando Σ={0,1} represente as seguintes linguagens com expressões regulares:**

{u∈Σ*: u é múltiplo de 4 mas não de 8} -(1(1|0)*)?100 || **{u∈Σ*: u tem comprimento 3}** - (0|1){3}

Automátos Finitos

Classificação de Autónamos: Um autómato finito diz-se ***determinístico*** (***AFD***) se, em cada um dos seus estados e perante um símbolo, puder transitar para um único estado. Caso contrário, diz-***se não determinístico (AFN)***. Há ainda autómatos finitos com transições vazias (AFε) em que é possível transitar de estado sem usar nenhum símbolo de Σ.São usados por exemplo, na conversão de Expressões Regulares em Autómatos Finitos. Comparação entre AFD e AFN: Os AFD são mais rápidos (tempo de computação) que os AFN; Os AFN ocupam muito menos espaço (têm menos estados) que os AFD;

$$A = (S, \Sigma, s_0, F, \delta)$$

- S é um conjunto finito de estados não vazio
- Σ é o alfabeto de entrada
- s_θ é o estado inicial
- F é o conjunto de estados finais
- δ é a função de transição
  - Recebe como argumentos
    - um estado
    - um símbolo de entrada
  - Devolve um novo estado

Exemplo: AF capaz de processar números binários terminados em `10`

Expressão regular: `(0|1)*10`

❖ Tabela de transições

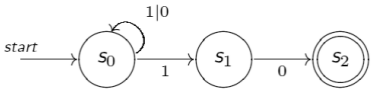
	0	1
→s_0	{s_0}	{s_0, s_1}
s_1	{s_2}	∅
*s_2	∅	∅

Autómato Finito:

$$A = (\{s_0, s_2, s_2\}, \{0,1\}, s_0, \{s_2\}, \delta)$$

$$\delta(s_0, 0) = \{s_0\}$$
$$\delta(s_0, 1) = \{s_0, s_1\}$$
$$\delta(s_1, 0) = \{s_2\}$$

❖ Representação gráfica



Gramatica:

Tipo3 - Gramáticas regulares. A produções são da forma: A → a || A → aB || A → ε em que A e B são dois quaisquer não terminais singulares e a é um qualquer terminal singular. Estas são as formas de gramáticas mais restritas em termos de poder de representação. **Tipo2 - Gramáticas independentes do contexto**. A produções são da forma: A → α em que α é uma sequência arbitrária de símbolos terminais e não terminais, sendo A um qualquer não terminal singular. Tal significa que qualquer ocorrência de A pode ser substituída por α independentemente do contexto. (2º imagem)

Representação Formal

```
G=({S}, {a,b}, P, S)
P: {S → aS
    S → b }

L(G)={aⁿb | n≥0}
```

```
G=({S,A,B}, {a,b}, P, S)
P: {S  → aB | bA
    A  → a | aS | bAA
    B  → b | bS | aBB }
```

Tipo1 - Gramáticas dependentes do contexto. A produções são da forma: αAβ → αγβ em que α,β e γ são sequências arbitrárias de símbolos terminais e não terminais, sendo que γ não é nulo e A é um qualquer não terminal singular. Estas gramáticas têm que respeitar a condição |αAβ| ≤ |αγβ| (img)**Tipo0 - Gramáticas livres** ou sem restrições. A produções são da forma:α → β em que tanto α como β são sequências arbitrárias de símbolos terminais e não terminais. O lado esquerdo da produção não pode ser vazio

```
G=({A,B,C}, {a,b,c}, P, A)
P: {A  → abc
    A  → aBbc
    Bb → bB
    Bc → Cbcc
    bC → Cb
    aC → aaB
    aC → aa }
```

Minimização de AFDs

	0	1
→ s_0	s_0	s_1
s_1	s_2	s_1
*s_2	s_0	s_3
s_3	s_2	s_3

		0	1
	→ s_0	s_0	s_1
A	s_1	s_2	s_1
	s_3	s_2	s_3
B	*s_2	s_0	s_3

		0	1
	→ s_0	s_0 (A)	s_1 (A)
A	s_1	s_2 (B)	s_1 (A)
	s_3	s_2 (B)	s_3 (A)
B	*s_2	s_0	s_3

	0	1
→ s_0	s_0	s_1
s_1	s_2	s_1
s_3	s_2	s_3
*s_2	s_0	s_3

		0	1
A	→ s_0	s_0	s_1
B	s_1	s_2	s_1
	s_3	s_2	s_3
C	*s_2	s_0	s_3

		0	1
A	→ s_0	s_0	s_1
B	s_1	s_2 (C)	s_1 (B)
	~~s_3~~	~~s_2~~ (C)	~~s_3~~ (B)
C	*s_2	s_0	~~s_3~~ s_1

		0		1	
NF	→A	A	NF	B	NF
	B	C	F	B	NF
	D	F	F	B	NF
	E	E	NF	D	NF
F	*C	E	NF	D	NF
	*F	A	NF	B	NF

		0		1	
G	→A	A	G	B	H
	E	E	G	D	H
H	D	F	F	B	H
	B	C	F	B	H
F	*C	E	G	D	H
	*F	A	G	B	H

A-Não Final(NF)
B-Final (F)

Conversão formal de AFNs em AFDs (simplificada)

1. Copiar estado inicial 2. Sempre que aparecer novos conjuntos , criá-los na tabela4. Criar um nome para cada estado
5. Substituir nomes nas transições6. Eliminar nomes antigos

Imagem (Conversão formal de AFNs em AFDs (simplificada))

	$\{A, a\}$	$\Sigma\backslash_{\{A,a\}}$
→ s_0	$\{s_0, s_1\}$	$\{s_0\}$
s_1	$\{s_2\}$	$\emptyset$
* s_2	$\{s_2\}$	$\{s_2\}$

	$\{A, a\}$	$\Sigma\backslash_{\{A,a\}}$
→ $\{s_0\}$ A	$\{s_0, s_1\}$B	$\{s_0\}$A
$\{s_0, s_1\}$ B	$\{s_0, s_1, s_2\}$C	$\{s_0\}$A
* $\{s_0, s1, s_2\}$ C	$\{s_0, s_1, s_2\}$C	$\{s_0, s_2\}$D
* $\{s_0, s_2\}$ D	$\{s_0, s_1, s_2\}$C	$\{s_0, s_2\}$D

	$\{A, a\}$	$\Sigma\backslash_{\{A,a\}}$
→ A	B	A
B	C	A
*C	C	D
*D	C	D

Representação formal

$$G = (V, \Sigma, P, S)$$

- V é um conjunto finito, não vazio, de símbolos **não terminais**
- Σ é o alfabeto (conjunto finito, não vazio, de **símbolos terminais**)
- P é o conjunto de **produções** (regras da gramática)
- S é o **símbolo inicial** a partir do qual todas as frases são derivadas (**axioma da gramática**)

Eliminação de sub-expressões comuns-Torna-se necessário examinar todas as instruções que podem ocorrer entre as duas avaliações da expressão. Se estas duas instruções não estão no mesmo bloco será necessário verificar quais outros blocos podem ser executados, para garantir que os valores que nosinteressam não são alterados.

Optimização Peephole-A geração de código objecto a partir de cada instrução individual do código fonte pode originar programas que possuam redundância e construções não optimizadas; Peephole é uma pequena janela que vai sendo deslocada ao longo do código O Se esse conjunto de instruções coincide com sequências pré-definidas para as quais haja um equivalente absoluto mais curto ou mais rápido faz-se a substituição

Polimorfismo- Funções polimórficas – funções em que as instruções do seu corpo podem ser executadas com argumentos de diferentes tipos; Funções polimórficas – facilitam a implementação de algoritmos que manipulem estruturas de dados, independentemente do seu tipo

O ***analisador semântico*** deve verificar se as construções estão correctas do ponto de vista semântico da linguagem. Assim as verificações realizadas pelo analisador semântico podem ser classificadas em: 1. verificação de tipos 2. verificação do fluxo de controle 3. verificação de unicidade 4. verificação relacionada aos nomes

Implementação dos parsers Ascendentes-Baseiam-se nos seguintes tipos de acções efectuadas sobre os tokens da entrada e sobreuma stack auxiliar (de terminais e não-terminais): *Deslocar (shift)*- retira da entrada o token corrente e coloca-o na stack (push)| *Reduzir (reduce)*- Substitui símbolos do topo da stack por um não-terminal, poraplicação de uma regra gramatical|*Aceitação* – termina a análise sintáctica reconhecendo a entrada. O texto da entrada é aceite se após o seu consumo estiver no topo da stack apenas o símbolo inicial da gramática|*Erro* – termina a análise sintáctica com erro de reconhecimento.

esquema XSD: schema

XML(1)

```
<?xml version="1.0"?>
<TotoBola concurso="21/2009">
<LinhaJogo>
<Jogo>Porto - Braga</Jogo>
<Aposta Escolha="1"/>
<Aposta Escolha="X"/>
</LinhaJogo>
<LinhaJogo>
<Jogo>Benfica - Belenenses</Jogo>
<Aposta Escolha="1"/>
</LinhaJogo>
<LinhaJogo>
<Jogo>P.Ferreira - Trofense</Jogo>
<Aposta Escolha="1"/>
<Aposta Escolha="X"/>
<Aposta Escolha="2"/>
</LinhaJogo>...
</TotoBola>
```

Operador	Descrição
|	Alternativa (ou)
+	Adição
-	Subtração
*	Multiplicação
div	Divisão
=	Igual
!=	Diferente
<	Menor
<=	Menor ou igual
>	Maior
>=	Maior ou igual
or	Disjunção (OU lógico)
and	Conjunção (E lógico)
mod	Módulo (resto da divisão)

/	Representa a raiz ou uma relação pai/filho
.	Representa o nó atual
..	Representa o pai do nó atual
@	Acesso a um atributo (@nomeatributo)
*	Representa qualquer nó
//	Todos os nós descendentes de um dado nó

Outras Tag:

```
<xsd:atribute name="n" type="xsd:string" fixed="pt-PT">
<xsd:atribute name="n" type="xsd:string" default="pt-PT">
<xsd:attribute name="n" type="T" use="required"/>
<xsd:attribute name="n" type="T" use="optional"/>
<xsd:attribute name="n" type ="T" use="prohibited"/>
```

Análise léxica -O Separa o texto de entrada numa sequência de tokens;O Detecta textos de entrada com tokens ilegais (erros léxicos);
O Análise sintáctica-O Aplica regras ramaticais;
O Constrói (por vezes implicitamente) uma árvore de parse com os tokens de entrada;O Detecta textos com sequências ilegais de tokens (erros sintácticos)

(XSD(1))

```
<?xml version="1.0"?>
<xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:simpleType name="Tescolha">
 <xsd:restriction base="xsd:string">
 <xsd:enumeration value="1"/>
 <xsd:enumeration value="X"/>
 <xsd:enumeration value="2"/>
 </xsd:restriction>
 </xsd:simpleType>
```

```
 <xsd:simpleType name="Tconcurso">
<xsd:restriction base="xsd:string">
<xsd:pattern value="[0-9]{2}/[0-9]{4}"/>
</xsd:restriction>
</xsd:simpleType>
```

```
<xsd:complexType name="Taposta">
<xsd:attribute name="Escolha" type="Tescolha"
use="required"/>
</xsd:complexType>
```

```
<xsd:complexType name="TLJ">
<xsd:sequence>
<xsd:element name="Jogo" type="xsd:string"/>
<xsd:element name="Aposta" type="Taposta"
maxOccurs="3"/>
</xsd:sequence>
</xsd:complexType>
```

```
<xsd:complexType name="Ttotobola">
 <xsd:sequence>
    <xsd:element name="LinhaJogo" type="TLJ"
minOccurs="13"  maxOccurs="13"/>
    </xsd:sequence>
   <xsd:attribute name="concurso"
type="Tconcurso" use="required"/>
 </xsd:complexType>
```

```
 <xsd:element name="TotoBola"
type="Ttotobola"/>
</xsd:schema >
```

Descida recursiva *ODesvantagens*-O não é geral, ou seja, os procedimentos são específicos para cada gramática;O tempo de análise é maior;O necessidade de uma linguagem que permita recursividade para sua implementação;O Dificuldade de validação| *OVantagens*-O simplicidade de implementação;O facilidade para inserir as diferentes funções do processo de compilação nomeadamente a possibilidade de inclusão de rotinas de análise;semântica – se existem diferentes alternativas para reescrever o mesmo não terminal.O eficiência – não será necessário tentar diferentes rotinas para oreconhecimento de uma frase (backtraking)

O ***tratamento de erros*** durante a análise sintáctica deve obedecer a certos princípios gerais: O O erro deve ser declarado o mais cedo possível, de modo a que possa ser localizado com precisão na sequência de tokens da entrada; O Depois de detectado um erro o parser deve continuar, num ponto mais à frente, de modo a detectar o maior número possível de erros numa única passagem;O parser deve tentar evitar o problema da "cascata de erros", em que um único erro gera uma série de mensagens;O parser deve evitar um ciclo infinito, reatando a análise sem consumir nenhum token,e gerando sucessivamente as mesmas mensagens de erros.

Árvores de parse- São representações gráficas (em forma de árvore) para as derivações, em que os nós dessa árvore têm associados símbolos terminais ou não-terminais da gramática | A raiz da árvore está sempre associada ao símbolo inicial| Os nós internos da árvores têm sempre associadossímbolos não-terminais|As folhas representam sempre terminais| Os filhos de um determinado nó interno representam a reescrita do não-terminal associado a esse nó, através de uma regra gramatical, num dos passos da derivação

Token – representa um conjunto de cadeias de entrada possível | **Lexema** –é uma determinada cadeia de entrada associada a um token

XSLT

<table>
<tr>
<td>
<xsl:template match="/" >
<html><body><h3>Lista de autores </h3>
<xsl:for-each
select="catalog/book/authors/author">
<xsl:sort select="."data-
type="text"order="ascending" />
<xsl:value-of
select="." /></br>
</xsl:for-each> </body></html>
</xsl:template>
</td>
<td rowspan="3">
Exemplo xsl:
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" version="1.0" encoding=
"ISO8859-1" indent="yes" />
<xsl:template match="/movies">
<html><head><title>Lista de Filmes</title></head><body>
<tableealign="center"border="1"><tr><th>Realizador</th>
<th>Filmes</th></tr>
<xsl:apply-templates select="directorfilms"/>
</table></body></html>
</xsl:template>
<xsl:template match="directorfilms">
<tr><td><xsl:value-of select="director/dirname"/></td>
<td><table>
<xsl:apply-templates select="films/film"/>
</table></td></tr>
</xsl:template>
<xsl:template match="film">
<tr><td><xsl:value-of select="t"/></td></tr>
</xsl:template>
</xsl:stylesheet>
</td>
</tr>
<tr>
<td>
<xsl:template match="book" >
<xsl:choose>
<xsl:when test="price < 5">
<xsl:value-of
select="title"/>
</xsl:when>
<xsl:otherwise> <xsl:value-of select="title"/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>
</td>
</tr>
<tr>
<td>
<xsl:template match="authors/author" >
<xsl:apply-templates />
<xsl:if test="position() !=last()"> ,</xsl:if>
</xsl:template>
</td>
</tr>
<tr>
<td>
<xsl:template match="Planta">
<tr><th><xsl:value-of select="position()"/></th>
<td><xsl:value-of select="NomeComum"/></td>
<td><xsl:value-of select="Preco"/></td>
<td><xsl:value-of select="Disponibilidade"/></td>
<td><xsl:element name="img">
<xsl:attribute name="src">
<xsl:apply-templates select="@NecessidadeLuz"/>
</xsl:attribute>
</xsl:element></td></tr></xsl:template>
</td>
<td>
<xsl:template match="@NecessidadeLuz">
<xsl:choose>
<xsl:when test=".='Sombra'">sombra.jpg</xsl:when>
<xsl:when test=".='Sol'">sol.jpg</xsl:when>
<xsl:when test=".='Preferencialmente
Sombra'">prefsombra.jpg</xsl:when>
<xsl:when test=".='Preferencialmente
Sol'">prefsol.jpg</xsl:when>
</xsl:choose>
</xsl:template>
</td>
</tr>
</table>

a(a|b)*b

Escreva uma sequência de derivação para aaabb.

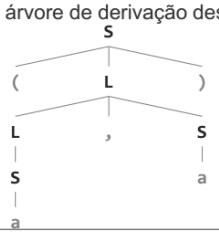
S → (L)|a
L → L,S|S

<S> ⇒ a<S> ⇒ aa<S> ⇒ aa<S>b ⇒ aaabb

a) Identifique os símbolos terminais e não terminais desta gramática

Símbolos terminais:	"a" "," "(" ")"
Símbolos não terminais:	S L

b) Determine uma árvore de derivação desta gramática para (a,a)



expressões XPA

```
<xsl:variable name="dataAtual"
select="fn:string(current-date())"/>
```

1) Obter o nome comum da segunda planta;
   `/Catalogo/Planta[2]/NomeComum`
2) Obter a quantidade de plantas no documento;
   `count(/Catalogo/Planta)`
3) Obter o preço da última planta da lista;
   `/Catalogo/Planta[last()]/Preco`
4) A média dos preços das plantas;
   `sum(/Catalogo/Planta/Preco) div count(/Catalogo/Planta)`
5) Formatar a média para duas casas decimais.
   `format-number(sum(/Catalogo/Planta/Preco) div count(/Catalogo/Planta),"#.00")`

Optimização de Código-O Um optimizador de código pode, opcionalmente, estar presente para melhorar o código (intermédio ou objecto) tanto em termos de velocidade, de espaço, ou ambosO A optimização automática pode ser feita tipicamente em três ocasiões:1. na representação intermediária, antes da geração de código|2. durante o processo de geração de código, que já é gerado de forma optimizada|3. após a geração de código directamente no código objecto ||três níveis de optimização: O Optimização local - aplica-se apenas a um bloco básico
|O Optimização global - aplica-se a todos os blocos básicos de um procedimento oufunção
|O Optimização inter-procedimental - aplica-se a todos os procedimentos de uma unidade de compilação

Vantagens do código intermédio: O **Reutilização**: Na construção de um compilador de uma dada linguagem para várias máquinas, só é necessário mudar o gerador final|O **Optimização**: O módulo de optimização pode ser o mesmo para vários compiladores de linguagens fonte diferentes e para máquinas alvo diferentes