Selectores de nós:

/	representa a raíz ou uma relação pai/filho
.	representa o nó corrente
..	representa o nó pai do nó corrente
@	modo de aceder a atributos *e.g.*, @nomedoatributo
*	selecciona todos os elementos filhos do nó actual
@*	seleciona todos os atributos do nó actual
//	todos os nós descendentes de um nó ou no princípio de um caminho todos os descendentes da raíz
:	separador de namespace
::	separador de eixo de navegação
text()	selecciona nós que contêm texto
node()	selecciona nós de qualquer tipo
comment()	selecciona nós comentário
processing-instruction('tipo')	selecciona nós instrução de processamento, se tipo existir só selecciona instruções desse tipo

Exemplos de selecção de nós:

../../title	selecciona o elemento `title` filho do avô (pai do pai) do nó corrente
//author	selecciona todos os nós `author` independentemente da sua posição no documento
/catalog/book/@id	selecciona o atributo `id` do nó `book`, filho de `catalog` debaixo da `raíz`
/catalog/*/title	selecciona todos os elementos `title` descendentes de todos os nós filhos de `catalog`
./author	selecciona todos os nós `author` filhos do nó actual
//@*	selecciona todos os atributos do documento
.//author	selecciona todos os elementos `author` descendentes do nó actual
//processing-instruction('php')	selecciona todos os nós instrução de processamento do tipo `php`

Operadores *XPath*

Operador	Descrição	Exemplo
|	Alternativa (OU)	//book | //cd
+	Adição	6 + 4
-	Subtracção	6 - 4
*	Multiplicação	6 * 4
div	Divisão	8 div 4
mod	Módulo (Resto da divisão Inteira)	5 mod 2
=	Igual	price = 9.80
!=	Diferente	price != 9.80
< (<)†	Menor	price < 9.80
<= (<)†	Menor ou igual a	price <= 9.80
> (>)†	Maior	price > 9.80
>= (>=)†	Maior ou igual a	price >= 9.80
or	Ou lógico	price=9.80 or price=9.70
and	E lógico	price > 9.00 and price < 9.90

† Dentro de um documento *XSLT*, devem ser usadas as entidades; num query com javascript DOM ou numa pesquisa *XPath* (ex. IntelliJ, netbeans, XML CopyEditor, Notepat++, etc.) usa-se o símbolo

Funções *XPath*[1]

`number position()`
devolve a posição do nó actual *e.g.*, `book[position()=1]`

`number last()`
devolve o índice do último nó no conjunto de nós *e.g.*, `book[last()]`

`number count(node-set)`
devolve o número de nós do conjunto de nós argumento *e.g.*, `count(//authors/author)`

`sum(node-set)`
Soma os valores seleccionados com o query *XPath e.g.*, `sum(price)`

`contains(string, substring)`
devolve true se a substring existe na string *e.g.*, `contains(title, "XML")`

Eixos de navegação *XPath*

child	Selecciona os filhos do nó corrente, eixo por omissão, qualquer nó excepto atributo ou namespace
parent	Selecciona o nó pai do nó corrente, se existir, equivalente a ..
self	Selecciona o nó corrente, self::* equivalente a .
attribute	Selecciona os atributos do nó corrente, equivalente a @
ancestor	Nós ascendentes até à raiz; pai, avó, etc.
ancestor-or-self	Ascendentes incluindo o próprio nó
descendant	Descendentes do nó actual: filhos, netos, etc.
descendant-or-self	Descendentes incluindo o próprio nó
preceding-sibling	Irmãos anteriores (todos os filhos do mesmo pai) ao nó actual, se o nó for atributo ou namespace devolve uma lista vazia
following-sibling	Irmãos seguintes (todos os filhos do mesmo pai) ao nó actual, se o nó for atributo ou namespace devolve uma lista vazia
preceding	Selecciona todos os nós que aparecem antes no documento e não são ascendentes
following	Selecciona todos os nós que aparecem depois no documento e não são descendentes
namespace	Selecciona nós namespace do nó actual

Declaração como conteúdo do elemento

```
<xsl:variable name="header">
  <tr>
    <th>Element</th>
    <th>Description</th>
  </tr>
</xsl:variable>
```

Utilização da variável

```
<xsl:value-of select="$header"/>
```

se for usada como conteúdo de um atributo podem-se usar { } para substituir `<xsl:value-of …>` , que não pode ser usado

```
<img src="{$link}" …
```

as { } também são válidas para qualquer expressão XPath

```
<xsl:template match="/" >
<html><body><h3>Lista de livros </h3>
  <xsl:apply-templates select="catalog/book">
    <xsl:sort select="pubdate" data-type="number"
              order="ascending"/>
    <xsl:sort select="title" data-type="text"
              order="ascending"/>
  </xsl:apply-templates>
</body></html>
</xsl:template>
```

Regioes existentes: `<xsl:apply-templates select="/ns:Catalogo/ns:Garrafa/ns:InfoAdicional/ns:Producao/ns:Regiao[not(.=../../../preceding-sibling::ns:Garrafa/ns:InfoAdicional/ns:Producao/ns:Regiao)]"/>`

Tipos de bebidas: `<xsl:value-of select="count(/ns:Catalogo/ns:Garrafa/ns:Tipo[not(.=../preceding-sibling::ns:Garrafa/ns:Tipo)])"/>`

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="html"/>
    <xsl:template match="/">
        <html>
            <head>
                <title>projetos.xsl</title>
            </head>
            <body>
                <p>Projetos 3+:  <xsl:value-of select="count(/Gestao/Projetos/Projeto[count(.//Elemento) &gt; 3])"/></p>
                <table>
                    <tr>
                        <th>Nome</th>
                        <th>Projectos</th>
                    </tr>
                    <xsl:apply-templates select="//Pessoa[@id = //Elementos/*/@id]">
                        <xsl:sort select="@nome" order="ascending"/>
                    </xsl:apply-templates>
                </table>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="//Pessoa">
        <xsl:variable name="mid" select="@id"/>
        <tr>
            <td>
                <xsl:value-of select="@nome"/>
            </td>
            <td>
                <xsl:value-of select="count(/Gestao/Projetos/Projeto[.//Elemento/@id = $mid])"/>
            </td>
        </tr>
    </xsl:template>
</xsl:stylesheet>
```

Parsers em descida recursivo

A descida recursiva pode ser usada, para implementar *parsers* preditivos;

Um *parser* preditivo é capaz de escolher a produção a aplicar simplesmente sabendo o **não terminal** actual e o **terminal** a ser processado;

Este tipo de gramáticas são chamadas de LL(1), onde:

- o primeiro "L" indica que a frase é processada da esquerda para a direita;
- o segundo "L" indica que se usa a derivação mais à esquerda;
- o (1) indica o número de terminais de avanço necessários para escolher entre produções alternativas;

Todas as gramáticas do tipo 3 podem ser convertidas em LL(1).

Formalmente, uma gramática é definida pelo tuplo $G = (V, \Sigma, P, S)$, no qual:

- V – um conjunto finito, não vazio, de **variáveis** (símbolos **não terminais**);
- Σ – é um conjunto finito, não vazio, dito **alfabeto** ou conjunto de símbolos **terminais**;
- P – conjunto de **produções**, regras da gramática. A sua forma geral é a seguinte: $\alpha \rightarrow \beta$ que definem a forma como o conjunto de símbolos α podem ser substituídos pelo conjunto de símbolos β;
- S – **símbolo inicial** a partir do qual todas as frases são derivadas.

Gramáticas do tipo 0

Gramáticas livres ou sem restrições. A produções são da forma:

$$\alpha \rightarrow \beta$$

em que tanto α como β são sequências arbitrárias de símbolos terminais e não terminais.

O lado esquerdo da produção não pode ser vazio.

Reconhece frases do tipo $\{a^n b^n c^n : n \geq 1\}$

Exemplo

$S \rightarrow aSBC|abC$
$CB \rightarrow BC$
$bB \rightarrow bb$
$bC \rightarrow bc$
$cC \rightarrow cc$

Gramáticas do tipo 1

Gramáticas dependentes do contexto. A produções são da forma:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

em que α,β e γ são sequências arbitrárias de símbolos terminais e não cterminais, sendo que γ não é nulo e A é um qualquer não terminal singular.

Estas gramáticas têm que respeitar a condição $|\alpha A \beta| \leq |\alpha \gamma \beta|$

A única excepção à regra anterior ($|\alpha A \beta| \leq |\alpha \gamma \beta|$) é a produção inicial ser $S \rightarrow \varepsilon$, mas **se e só se** o S não existir do lado direito (para permitir a palavra vazia).

Reconhece frases do tipo $\{a^n b^n c^n : n \geq 1\}$

Exemplo

$S \rightarrow aSBC|abC$
$CB \rightarrow XB$
$XB \rightarrow XC$
$XC \rightarrow BC$
$bB \rightarrow bb$
$bC \rightarrow bc$
$cC \rightarrow cc$

Gramáticas do tipo 2

Gramáticas independentes do contexto. A produções são da forma:

$$A \rightarrow \alpha$$

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.

Reconhece frases do tipo $\{a^n b^p d^q c^n : n, p, q \geq 0\}$

Exemplo

$A \rightarrow aAc$
$A \rightarrow BD$
$B \rightarrow bB|\varepsilon$
$D \rightarrow dD|\varepsilon$

Gramáticas do tipo 3

Gramáticas regulares. A produções são da forma:

$$A \rightarrow a$$
$$A \rightarrow aB$$
$$A \rightarrow \varepsilon$$

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.

Lineares à direita	Lineares à esquerda
$A \rightarrow a$	$A \rightarrow a$
$A \rightarrow aB$	$A \rightarrow Ba$
$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$

Nota: A produção $A \rightarrow a$ pode ser omitida, pois pode ser derivada a partir das outras duas.

Atributos: valores associados aos símbolos da gramática; Acções

Semanticas: definem dependências entre atributos ;Atributos

Sintetizados: que dependam dos atributos dos filhos numa árvore de parse; Atributos herdados: dependência entre atributos de símbolos do lado direito das regras, ou de atributos do símbolo não terminal do lado esquerdo.

Código Fonte é uma linguagem de:

De alto nivel (p. e., algorítmicas, imperativas, procedimentais)

Código maquina é uma linguagem de:

Baixo nivel

O teste de Turing ["The Imitation Game"] permite:

Distinguir se a máquina do teste comporta-se como uma máquina ou um ser humano, ou seja, se é inteligente ou não.

Turing é considerado:

O fundador da computação moderna.Primeiro modelo teórico para o computador universal.

A máquina de Turing reconhece linguagens:

Do tipo zero. Primeiro dispositivo básico para a manipulação de símbolos simples, já com conceitos básicos de memória, programa e linguagem de programação

Um compilador é um programa capaz de:

Interpretar um ficheiro (programa numa determinada linguagem de programação; linguagem fonte) e transforma-lo num programa em linguagem equivalente ou superior (linguagem alvo), identificando a presença de erros no código fonte.

O Assembler é um tradutor de:

Linguagem de baixo nível (de instrução de linguagem para instrução maquina)

Os compiladores e interpretadores são tradutores de:

Linguagens de alto nível (de instrução de linguagem para várias instruções maquina)

A principal diferença em um compilador e um interpretador é que:

Compilador traduz código fonte integralmente em linguagem máquina para o programa ser executado como de uma vez

Interpretador traduz da mesma forma mas executa instrução a instrução até o programa terminar

Uma gramática é composta por 4 componentes:

Símbolos terminais (tokens), símbolos não terminais (gramáticas), produções (regras de gramática) e símbolo inicial (símbolo de partida).Uma ambigua nunca pode ser LR(1)

Existem duas classes num analisador sintático: Analise top-down (árvore de parse construída a partir da raiz até atingir todos os terminais) e Analise bottom-up (árvore de parse construída a partir das folhas até atingir o símbolo inicial).

Um token representa: Um conjunto de cadeias e entrada possível

Um lexema é: Uma cadeia de entrada associada a um token.

BISSON: É implementado através de um autómato, "shift"; ▸ Quando tem no topo da pilha os símbolos do lado direito de uma regra faz "reduce", isto é, transforma esses símbolos no não terminal (regra) que fica no topo da pilha. Quando há necessidade de mais de um tipo pode ser usada a definição de uma %union; ywhalguardar os valores semânticos (lexemas). A precedência de operadores serve para definir a ordem pela qual as regras alternativas são processadas; eliminando assim os conflitos que possam surgir; pode analisar gramáticas com recursividade à direita ou à esquerda(mais eficientes).**CONFLITOS:** Quando o BISON pode realizar "reduce" a 2 regras simultâneamente gera um conflito reduce/reduce; "shift" de um token ou "reduce" a 1 regras simultâneamente gera um conflito shift/reduce.Escolhe o Shift; Estes erros surgem normalmente da ambiguidade da gramática.**RECUPERAÇÃO ERROS:** continuar a analise mesmo quando encontra um erro; Pode ser realizada com regras que "simulem" o erro; Pode ser realizada usando o "token" error que instância com qualquer sequência de "tokens". O ANTLR gera um analisador parsico top down enquanto o bisson gera um analisador parsico bottom up. O ANTLR gera parsers descendentes recursivos, cuja logica de processamento está no codigo do parser, são geralmente mais rapidos do que os parsers de tabela. O BISON gera um parser cuja logica de processamento esta contida na informação do programa e nao no codigo do parser, tendo uma footprint de memoria inferior.

ANALISADOR LEXICO

O analisador lexico(gerado por bison ou yacc) funciona como uma subrotina do analisador sintático. Ao construir um analisador lexico deve começar-se por escrever uma expressão regular que entenda a definição de todos os tokens da linguagem. **FUNÇÕES:**-ignorar alguns tipos de caracteres;-mensagens de erro quando algum caracter não é reconhecido;-localizar os tokens no texto fonte;-atuar como pré-processador.

ANALISADOR SINTATICO: verificar se uma dada sequência de tokens constitui um programa valido, construindo a respectiva árvore de parse de acordo com as regras gramaticais dadas.Procurar erros na sequencia de tokens. **INPUT:**sequencia de tokens vindos do analisador lexico; regras recursivas;**OUTPUT:**arvore de parse segundo a gramatica.

Analise top-down(derivação mais à esquerda) (árvore de parse construida a partir da raiz até atingir todos os terminais) e Analise bottom-up(descida recursiva e analise preditiva não recursiva) (árvore de parse construida a partir das folhas até atingir o simbolo inicial). O conflito "reduce-reduce" verifica-se quando o analisador sintático ascendente não consegue decidir qual das diversas reduções alternativas realizar.Gramaticas tipo 2.TOP DOWN:tem-se uma stack de reconhecimento que pode conter sequências alternadas de variáveis, terminais ou estados. efectua as seguintes operações básicas sobre os tokens de entrada e sobre a stack: shift (retira da entrada o token corrente ecoloca-o na stack), reduce (substitui simbolos do topo da stack por uma não terminal e por aplicação de uma regra gramatical), aceitação (termina a analise sintática reconhecendo a entrada, tendo que estar, no topo da stack, o simbolo inicial) e erro (termina a analise sintática com erro de reconhecimento). Vantagens: sua simplicidade de implementar e a sua facilidade de inserir diferentes alternativas para reescrever o mesmo simbolo não terminal.

Desvantagens:maior tempo de analise;dificuldade de validação

ANALISADOR SEMANTICO:

É usada para verificar se as regras semânticas da linguagem são satisfeitas e obter o significado completo da frase ao determinar os valores dos símbolos. **VERIFICAÇÕES:** Verificação de tipos, verificação do fluxo de controlo, verificação de unicidade, verificação relacionada aos nomes.

VERIFICAÇÃO DE TIPOS: As tabelas de símbolos são usadas nos compiladores:nas ocorrências definidoras de identificadores e nas ocorrências aplicadas.

ERROS: A detecção e recuperação de erros:deve detectar os erros ocorridos, avisar o utilizador das ocorrências e recuperar de modo a poder analisar o código fonte até ao fim.**GERAÇÃO DE CODIGO:** O código intermédio:é independente da máquina alvo.A geração de código no modelo frontend-backend inclui a verificação de tipos.O módulo de front-end de um compilador é independente da máquina alvo.

Derivação canónica é um método sistemático para executar uma derivação: duas formas de o fazer: ou escolher sempre, para reescrita, o não terminal mais esquerda ou, alternativamente, escolher sempre, para reescrita, o não-terminal mais à direita;**OPTIMIZAÇÃO:** pode ser feita em três ocasiões: na representação intermédia,durante o processo de geração de código ou directamente sobre o código objecto. Pode ocorrer a três níveis: local (onde actuam a maior dos compiladores), global ou inter-procedimental. Alterar algoritmos;transformar os ciclos;melhorar ciclos,chamadas a procedimentos e o cálculo de endereços;eliminar subexpressões comuns e código "morto".

Linguagens de Programação são: Linguagens formais utilizadas para descrever e comunicar um processo computacional.

Linguagens Formais são: Conjuntos de cadeias de símbolos de um alfabeto

A estrutura do compilador pode ser dividida em duas partes: Front end (vanguarda), que origina o código-fonte intermédio e reconheça código-fonte válido

Back end (retaguarda), que produz o código máquina a partir do código intermédio e assegura compatibilidade com os diferentes sistemas destino. simplifica a producao de compiladores

Pré-processador -> Compilador -> Assembler -> Linker -> Código fonte alvo

Gramáticas de atributos: define como o conjunto de atributos e regras semanticas que expressam como a computacao dos atributos se relacionam com as regras gramaticais da linguagem.(Falar sobre atributos sintetizados e herdados)

Enumeration	Define uma lista de valores válidos
fractionDigits	Especifico o número máximo de casas decimais permitidas. Deve ser maior ou igual que zero.
Length	Especifica o número exacto de caracteres ou itens permitidos. Deve ser maior ou igual que zero.
maxExclusive	Especifica o valor máximo para valores numéricos (o valor deve ser menor que este valor).
maxInclusive	Especifica o valor máximo para valores numéricos (o valor deve ser menor ou igual a este valor).
maxLength	Especifica o número máximo de caracteres ou itens permitidos. Deve ser maior ou igual que zero.
minExclusive	Especifica o valor mínimo para valores numéricos (o valor deve ser maior que este valor).
minInclusive	Especifica o valor mínimo para valores numéricos (o valor deve ser maior ou igual a este valor)
minLength	Especifica o número mínimo de caracteres ou itens permitidos. Deve ser maior ou igual que zero.
Pattern	Define a sequência exacta de caracteres permitidos
totalDigits	Especifica o número exacto de dígitos permitidos. Deve ser maior que zero
whiteSpace	Especifica como caracteres vazios (tabs, espaços e retornos de carro) são tratados

```
<xsd:element name="carro">
   <xsd:simpleType>
     <xsd:restriction base="xsd:string">
         <xsd:enumeration value="Audi"/>
         <xsd:enumeration value="Golf"/>
         <xsd:enumeration value="BMW"/>
     </xsd:restriction>
   </xsd:simpleType>
</xsd:element>

<xsd:element name="idade">
  <xsd:simpleType>
   <xsd:restriction base="xsd:integer">
       <xsd:minInclusive value="0"/>
       <xsd:maxInclusive value="100"/>
   </xsd:restriction>
  </xsd:simpleType>
</xsd:element>

<xsd:element name="letra">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[a-z]"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

- xsd:string
- xsd:decimal
- xsd:integer
- xsd:boolean
- xsd:date
- xsd:time

elementos alternativos (minOccurs= nonNegativeInteger e maxOccurs= nonNegativeInteger|unbounded). Por omissão, o valor destes

```
<xsd:element name="funcionário" type="pessoa"/>
<xsd:element name="estudante" type="pessoa"/>
<xsd:element name="membro" type="pessoa"/>
<xsd:complexType name="pessoa">
    <xsd:sequence>
        <xsd:element name="firstname" type="xsd:string"/>
        <xsd:element name="lastname" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="product">
  <xsd:complexType>
     <xsd:attribute name="prodid" type="xsd:positiveInteger"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="pessoas">
   <xsd:complexType>
      <xsd:sequence maxOccurs="10" minOccurs="0"/>
         <xsd:element name="full_name" type="xsd:string"/>
         <xsd:element name="child_name" type="xsd:string"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

```
<xs:element name="id" type="xs:ID">
O tipo xs:ID é usado para validar um identificador único e usado em
atributos ou elementos simples;
Não podem existir dois elementos/atributos do tipo xs:ID com o mesmo
identificador.
No código seguinte, o validador de XSD, dá erro no segundo <id> pois é
repetido:
<id>abcd</id>
<id>abcd</id>
O conteúdo do tipo xs:ID é uma palavra e tem que iniciar com uma letra.
<xs:element name="idAluno" type="xs:IDREF">
O tipo xs:IDREF é usado para validar atributos ou elementos que no
documento existam como xs:ID;
Se não existir no documento um xs:ID com o mesmo valor é gerado um erro
pelo validador XML.
Também existe o tipo xs:IDREFS que aceita uma lista de xs:ID separados
por espaços
```

```
<xsd:group name="persongroup">
     <xsd:sequence>
         <xsd:element name="firstname" type="xsd:string"/>
         <xsd:element name="lastname" type="xsd:string"/>
         <xsd:element name="birthday" type="xsd:date"/>
     </xsd:sequence>
</xsd:group>

<xsd:element name="person" type="personinfo"/>

<xsd:complexType name="personinfo">
  <xsd:sequence>
    <xsd:group ref="persongroup"/>
    <xsd:element name="country" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

- "**preserve**" - significa que o processador XML não vai remover nenhum carácter vazio
- "**replace**" - significa que o processador XML vai substituir todos os caracteres vazios (quebras de linha, tabs, espaços) por espaços.
- "**collapse**" - significa que o processador XML vai remover todos os caracteres vazios (quebras de linha, tabs, espaços são substituídos com espaços, espaços iniciais e finais são removidos, espaços múltiplos são reduzidos a um).

exemplo seguinte define um elemento chamado <endereço> com uma restrição:

```
<xsd:element name="endereço">
 <xsd:simpleType>
    <xsd:restriction base="xsd:string">
        <xsd:whiteSpace value="collapse"/>
    </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
```

```
<xsd:element name="password">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
         <xsd:length value="8"/>
      </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

.	any character except newline
\w \d \s	word, digit, whitespace
\W \D \S	not word, digit, whitespace
[abc]	any of a, b, or c
[^abc]	not a, b, or c
[a-g]	character between a & g
Anchors	
^abc$	start / end of the string
\b \B	word, not-word boundary
Escaped characters	
\. * \\	escaped special characters
\t \n \r	tab, linefeed, carriage return
\u00A9	unicode escaped ©
Groups & Lookaround	
(abc)	capture group
\1	backreference to group #1
(?:abc)	non-capturing group
(?=abc)	positive lookahead
(?!abc)	negative lookahead
Quantifiers & Alternation	
a* a+ a?	0 or more, 1 or more, 0 or 1
a{5} a{2,}	exactly five, two or more
a{1,3}	between one & three
a+? a{2,}?	match as few as possible
ab|cd	match ab or cd

AttributeGroup name - usados para definir grupos de atributos.

```
<xsd:attributeGroup name="personattrgroup">
     <xsd:attribute name="firstname" type="xsd:string"/>
     <xsd:attribute name="lastname" type="xsd:string"/>
     <xsd:attribute name="birthday" type="xsd:date"/>
</xsd:attributeGroup>
```

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
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	s_1
A	s_1	s_2	s_1
	s_3	s_2	s_3
B	*s_2	s_0	s_3
C	*s_2	s_0	s_3

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

Conversão formal de AFNs em AFDs (simplificada)

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

```
<xsd:element name="password">
    <xsd:simpleType>
     <xsd:restriction base="xsd:string">
         <xsd:minLength value="5"/>
         <xsd:maxLength value="8"/>
     </xsd:restriction>
    </xsd:simpleType>
</xsd:element>

<xsd:element name="person">
  <xsd:complexType>
     <xsd:all>
        <xsd:element name="firstname" type="xsd:string"/>
        <xsd:element name="lastname" type="xsd:string"/>
     </xsd:all>
  </xsd:complexType>
</xsd:element>

<xsd:element name="identificacao">
  <xsd:complexType>
     <xsd:choice>
        <xsd:element name="bi" type="BI"/>
        <xsd:element name="cc" type="CC"/>
     </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

```
<xsd:attribute name="lang" type="xsd:string" default="pt">
<xsd:attribute name="lang" type="xsd:string" fixed="pt"/>
<xsd:attribute name="lang" type="xsd:string" use="optional"/>
<xsd:attribute name="lang" type="xsd:string" use="required"/>
```

```
<?xml version="1.0"?>
<xs:schema version="1.0"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified" xmlns:zs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="Gestao" type="TGestao"/>
    <xs:complexType name="TGestao">
        <xs:sequence>
            <xs:element name="Projetos" type="TProjetos"/>
            <xs:element name="Pessoas" type="TPessoas"/>
        </xs:sequence>
    </xs:complexType>
    <!-- COMPLETAR schema -->
    <xs:complexType name="TProjetos">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="Projeto" type="TProjeto"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="TPessoas">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="Pessoa" type="TPessoa"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="TPessoa">
        <xs:attribute name="id" type="TPessoaID"/>
        <xs:attribute name="nome" type="xs:string"/>
        <xs:attribute name="classificacao" type="TPessoaClass" use="optional"/>
    </xs:complexType>

    <xs:complexType name="TProjeto">
        <xs:sequence>
            <xs:element name="Elementos" type="TElementos"/>
            <xs:element name="DataInicio" type="xs:date"/>
        </xs:sequence>
        <xs:attribute name="id" type="TProjetoID"/>
        <xs:attribute name="designacao" type="TProjetoDesignacao"/>
    </xs:complexType>

    <xs:complexType name="TElementos">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="Elemento" type="TElemento"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="TElemento">
        <xs:attribute name="id" type="xs:IDREF"/>
    </xs:complexType>
```

```
<xs:simpleType name="TProjetoID">
    <xs:restriction base="xs:ID">
        <xs:pattern value="p[0-9]{3}"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="TPessoaID">
    <xs:restriction base="xs:ID">
        <xs:pattern value="ps[0-9]{3}"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="TProjetoDesignacao">
    <xs:restriction base="xs:string">
        <xs:minLength value="5"/>
        <xs:maxLength value="30"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="TPessoaClass">
<xs:restriction base="xs:string">
    <xs:pattern value="Excelente|Muito Bom|Bom"/>
</xs:restriction>
</xs:simpleType>

</xs:schema>
```