XML e XSL
da Teoria à Prática

José Carlos Leite Ramalho Pedro Rangel Henriques

20 de Abril de 2001

Prefácio

Quando o projecto desta obra nasceu, ainda no ano de 2000, o objectivo era escrever um livro sobre XML. A ideia era a de criar um livro que pudesse servir ao curioso mas também ao estudante da área e utilizador mais experiente. A temática do XML cedo se revelou insuficiente. Um utilizador de XML, normalmente, quer fazer alguma coisa com os documentos que vai criando. Para isso, é necessário transformar os documentos XML. Assim, este livro cobre, exaustivamente, numa primeira parte, o XML e, numa segunda parte, a temática da transformação dos documentos.

Não é necessário voltar a reforçar aqui a importância do XML. Hoje, esta norma invadiu todas as áreas do saber onde a informática está presente. Domínios tão diversos como a medicina, a indústria aeronáutica, o xadrez, a matemática, a música, as letras e o teatro fazem parte das áreas de aplicação desta tecnologia. Daí a preocupação de fazer uma obra abrangente que possa dar resposta a quem trabalha em todas aquelas áreas.

O livro reflecte a experiência acumulada dos autores. Além de descrever os conceitos básicos, apresenta metodologias e *maneiras de fazer* que só se adquirem após vários anos de experiência. Um dos objectivos deste livro é também o de acelerar a curva de aprendisagem do XML que ainda é grande e apontada como umas das desvantagens da tecnologia.

Quando se escreve uma obra há, num determinado momento, a necessidade de colocar um ponto final e terminar. Foi o que fizemos mas, ficando com a sensação que

muito podia ainda ser escrito. No momento em que este livro estiver a ser impresso, estaremos já a trabalhar na próxima edição, para que esta dê mais e melhor.

Índice

Parte I

XML

1 Documentos XML bem formados

Como o leitor pode constatar, um documento XML é formado por dados e anotações.

As primeiras perguntas que podem surgir são: o que é que são anotações? Como distingui-las dos dados?

Num documento XML, os dados são blocos de texto. Pode ser que um dia os dados noutros formatos que não texto possam vir a fazer parte de um documento XML, mas neste momento são externos aos documentos, podendo ser referidos através de entidades (capítulo **??**).

A anotação de um documento descreve a sua estrutura e induz uma interpretação do seu conteúdo. A anotação é composta por: marcas de início de elementos, marcas de fim de elementos, marcas de elementos vazios, referências a entidades, comentários, limitadores de secções especiais de texto (CDATA – secção 1.6), declarações de tipo de documento e instruções de processamento.

Veja-se agora, representado em XML, o exemplo tradicionalmente usado noutros contextos.

Exemplo 1: "Hello World"

```
<?xml version="1.0" encoding="UTF-8"?>
<doc>
  Hello World!!!
</doc>
```

Uma anotação começa sempre por '<' e termina por '>'. Por isso é fácil ver que o nosso exemplo tem três anotações: `<?xml ...?>`, `<doc>` e `</doc>`.

1.1 A declaração XML

De acordo com a norma ([**?**]), um documento XML deve começar sempre com uma **declaração XML** (que tem estado presente em todos os exemplos fornecidos até agora). Normalmente, se o documento tiver algo antes da declaração ou esta estiver ausente, qualquer processador acusará um erro ao tentar processar o documento.

Uma declaração XML é uma anotação especial que tem a seguinte forma:

```
<?xml
    version="1.0"
    standalone="yes"
    encoding="iso-8859-1" ?>
```

Podem ser usados três atributos numa declaração XML:

version O valor deste atributo indica a versão de XML que está a ser utilizada. Neste momento, apenas o valor "1.0" é possível. Este atributo é obrigatório e tem de estar presente em todas as declarações.

standalone Este atributo é opcional e pode ter um de dois valores: o valor "yes" indica que o documento está autocontido, não tem referências a entidades externas; o valor "no"indica que o documento contém referências a entidades externas (por exemplo: outros documentos).

encoding Este atributo é também opcional e indica qual a codificação usada para os caracteres: o valor por omissão é UTF-8, mas também poderá ser UCS-2, UCS-4, ISO-XXX. No nosso caso, usamos caracteres portugueses, por isso, este atributo terá sempre o valor "iso-8859-1".

1.2 Comentários

Um **comentário** pode aparecer em qualquer ponto de um documento XML.

Começa pela marca `<!--` e termina com a marca `-->`.

Exemplo 2: "Hello World"

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!--Isto é um comentário no início-->
<doc>Olá Mundo!!!</doc>
```

Existem algumas restrições à utilização de comentários:

- Não podem aparecer antes da declaração.
- Não podem aparecer dentro de uma anotação.
- Não se pode utilizar a sequência de caracteres "`--`"dentro de um comentário.

Além da sua função óbvia de documentação, os comentários podem ainda ser utilizados para remover temporariamente partes do documento, desde que essas partes não contenham comentários.

Exemplo 3: Uso de comentários para remover partes de um documento

```
<RECEITAS>
  <TITULO> O Meu Livro de Receitas </TITULO>
  <RECEITA ORIGEM="Portugal">
    <TITULO> Bolo </TITULO>
 <!--
    <INGREDIENTE> 500g de farinha </INGREDIENTE>
 -->
    <INGREDIENTE> 200g de açucar </INGREDIENTE>
    <INGREDIENTE> 300g de manteiga </INGREDIENTE>
  </RECEITA>
</RECEITAS>
```

1.3 Instruções de processamento

As instruções de processamento são a única reminiscência da anotação procedimental que ainda perdura no XML. Uma instrução de processamento não faz parte do conteúdo do documento. É uma indicação directa ao processador do documento de que algo deve ser executado naquele ponto quando o documento estiver a ser transformado.

Uma instrução de processamento começa por `<?id-processador` e termina por `?>`; `id-processador` deverá indicar a que tipo de processamento a instrução se destina.

Note-se que a declaração XML (secção 1.1) não é mais do que uma instrução de processamento.

A seguir apresenta-se um documento XML que contém algumas instruções de processamento.

Exemplo 4: Instruções de processamento

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<agenda>
  <?html action="hr"?>
  <entrada id="e1" tipo="pessoa">
    <nome>José Carlos Ramalho</nome>
    <email>jcr@di.uminho.pt</email>
    <telefone>253 604479</telefone>
  </entrada>
  <?html action="hr"?>
```

Neste exemplo, além da já referida declaração XML, estão presentes duas instruções de processamento análogas:

<**?html ...?**> estas duas instruções de processamento identificam ordens que deverão ser executadas aquando da transformação deste documento para HTML. Neste caso concreto, indica-se que, quando o processamento do documento XML coincidir com a geração da sua versão HTML, deverão ser geradas duas marcas `hr` nas posições onde se encontram as instruções de processamento.

Neste momento, já vimos tudo o que pode aparecer no início de um documento XML (antes do corpo do documento): instruções de processamento, comentários e a declaração XML[1].

Um documento XML bem formado tem várias componentes que se descrevem nas secções seguintes.

1.4 Elementos

A estrutura de um documento XML é definida à custa de **elementos** que determinam os blocos lógicos em que o texto global pode ser partido ou decomposto. Um

[1] Embora os dois primeiros possam aparecer em qualquer ponto do corpo do documento.

elemento é composto pela sua anotação de início, pelo seu conteúdo e pela sua anotação de fim.

Uma anotação de início começa por '<' e termina com '>', e uma anotação de fim começa por '</' e termina com '>'. Quer uma quer outra contêm no meio o nome do elemento que inicia ou que termina, respectivamente.

Exemplo 5: Anotação de um elemento

No texto abaixo, a palavra `Braga` está anotada como sendo um elemento de nome `lugar`.

```
Vais ver o espectáculo a <lugar>Braga</lugar>?
```

Uma restrição inerente ao XML é a de que o nome de uma anotação de fim tem de ser igual ao da anotação de início imediatamente anterior (este será um dos princípios da *bem formação* de um documento).

Um elemento tem de estar completamente contido noutro elemento, excepto o ancestral de todos os elementos (o elemento raiz). Algumas estruturas hierárquicas podem ser recursivas. Um elemento pode conter directa ou indirectamente instâncias de si próprio, como iremos ver mais à frente num caso de estudo (exemplo 23). Esta possibilidade de recursividade ou aninhamento irá causar, muitas vezes, problemas no momento de processar a informação, mas é necessária para modelar certo tipo de informação.

As regras a que o nome de um elemento deve obedecer são as seguintes:

1. o primeiro carácter deverá ser uma letra, um *underscore* ou um sinal de dois pontos;
2. os caracteres seguintes podem ser letras, dígitos, *underscores*, hífens, pontos e dois pontos;
3. o espaço em branco não pode aparecer no nome de uma anotação.

Em XML há distinção entre maiúsculas e minúsculas. Assim, `lugar`, `Lugar` e `LUGAR` são referências a elementos distintos.

Eis alguns exemplos de nomes válidos:

```
<Doc:princ>       </Doc:princ>
<documento>       </documento>
<_secreto>        </_secreto>
<aluno4>          </aluno4>
<DB_tab5>         </DB_tab5>
```

E alguns exemplos de nomes inválidos:

```
<1documento>     </1documento> -- começa por dígito
<aluno(4)>       </aluno(4)>   --  tem parentesis
<DB tab5>        </DB tab5>    -- tem espaço em branco
```

Por outro lado, no conteúdo de um elemento, nunca deverão aparecer os caracteres '<' e '>' pois são os caracteres que limitam as anotações. Em lugar deles devem-se usar, respectivamente, as entidades (capítulo **??**) do tipo carácter '<' e '>'. Qualquer processador ou editor de XML fará a substituição automática daquelas entidades pelos caracteres correspondentes.

Exemplo 6: Caracteres reservados

Se, no documento, estivesse escrito desta forma:

```
 A anotação &lt;nome&gt; é usada para
      anotar nomes.
```

Um editor mostraria o mesmo texto da seguinte maneira:

```
A anotação <nome> é usada para anotar
      nomes.
```

1.4.1 Tipos de conteúdo

Os elementos podem incluir outros elementos (ditos elementos filho) e texto.

Há elementos que não contêm texto directamente, apenas contêm outros elementos. São elementos estruturantes. Num dos exemplos anteriores, apresentou-se o livro de receitas: o elemento RECEITAS tem como conteúdo apenas elementos RECEITA; é portanto um elemento deste tipo.

```
<RECEITAS>
    <RECEITA> ... </RECEITA>
    <RECEITA> ... </RECEITA>
    <RECEITA> ... </RECEITA>
    ...
</RECEITAS>
```

Os caracteres brancos (espaços, mudanças de linha, tabulações) que surgem no conteúdo de elementos que não contêm texto são irrelevantes e não são considerados como fazendo parte do documento. Assim, o excerto anterior poderia ser igualmente representado por:

```
<RECEITAS><RECEITA> ... </RECEITA><RECEITA> ...
</RECEITA><RECEITA> ...</RECEITA>...</RECEITAS>
```

No entanto, em algum ponto da hierarquia do documento, o texto irá aparecer. Neste ponto o elemento contém texto ou uma mistura de texto com elementos filho.

Um elemento que contenha apenas texto é designado como tendo **conteúdo textual**. São exemplo desta situação, os elementos abaixo, que já foram aparecendo neste livro:

```
<lugar>Braga</lugar>
<INGREDIENTE>Meia dúzia de ovos</INGREDIENTE>
<data>(1922)</data>
```

Quando um elemento contém simultaneamente texto e elementos filho, designa-se por elemento de **conteúdo misto**. Nos exemplos abaixo, `<verso>` e `<p>` são instâncias de elementos de conteúdo misto (que também já foram introduzidos anteriormente), enquanto `<nome>` e `<lugar>` são instâncias de elementos de conteúdo textual.

```
<verso>Olha, <nome>Daisy</nome>: quando ...</verso>
<p>Vais ver o espectáculo a <lugar>Braga</lugar>?</p>
```

Por último, um elemento pode não ter qualquer conteúdo. Este tipo de elemento é normalmente designado por elemento **vazio**.

Estes elementos são normalmente utilizados pelo seu significado posicional. Alguns exemplos conhecidos do HTML são os elementos BR e HR que marcam, respectivamente, a quebra de uma linha e o traçar de uma linha horizontal. O leitor aqui poderia interrogar-se: "Não será isto uma anotação procedimental?". Terá um pouco de razão, mas relembre a discussão sobre uma anotação equilibrada (**??**).

Há, no entanto, elementos vazios puramente descritivos como é o caso de elementos que representam referências. Para além da posição, contêm informação nos atributos (1.5). Eis um exemplo de um destes elementos:

```
Como será discutido num capítulo mais à
frente (<ref ident="cap5"/>) ...
```

Outro exemplo deste tipo de elementos, que aparecerá mais à frente nalguns casos de estudo, é o de um elemento que marca o ponto onde deverá ser inserida uma imagem.

```
Como se pode ver na figura seguinte ...
<figura path="..."/>
...
```

Como já deve ter notado, estes elementos têm uma sintaxe diferente. São representados por uma única anotação que é iniciada por '`<`' e termina em '`/>`', que é a forma abreviada de escrever `<elem-ident></elem-ident>`.

Na próxima secção, vamos analisar os atributos que surgiram nestes últimos exemplos.

1.5 Atributos

Um elemento pode ter um ou mais atributos que, por sua vez, podem ser opcionais ou obrigatórios. Os atributos visam qualificar o elemento a que estão associados.

Ao pensar no problema da distinção entre elemento e atributo, podemos traçar um paralelo com a língua portuguesa: os elementos são os substantivos, e os atributos os adjectivos. Podemos então estabelecer a seguinte equivalência:

```
Isto é uma casa        ---  <CASA>
Isto é uma casa verde  ---  <CASA COR="verde">
```

Não há limite para o número de atributos que podem estar associados a um elemento.

Os atributos aparecem sempre na anotação que marca o início de um elemento, uma vez que vão qualificar o conteúdo que se segue. Um atributo é definido por um par constituído por um nome e um valor: o nome e o valor devem estar separados pelo sinal = e o valor deverá estar colocado dentro de aspas simples ou duplas.

O nome de um atributo segue as mesmas regras do nome dos elementos. O valor será sempre o texto que estiver dentro das aspas.

Podemos desde já alertar o leitor para o problema da representação de uma dada informação num elemento ou num atributo. Não existe uma fronteira entre os dois e muitas vezes a escolha não é simples. Atente-se no seguinte exemplo de uma agenda de contactos.

Exemplo 7: Informação nos elementos

```
<agenda>
  <entrada id="e1" tipo="pessoa">
    <nome>José Carlos Ramalho</nome>
    <email>jcr@di.uminho.pt</email>
    <telefone>253 604479</telefone>
  </entrada>
  ...
</agenda>
```

Uma agenda é composta por um ou mais elementos `entrada`. Cada um destes elementos é estruturado em vários subelementos, `nome`, `email` e `telefone`, e tem dois atributos – `id` e `tipo` – que lhe atribuem certas propriedades. Vamos ver o que acontece se passarmos a informação dos elementos para atributos.

Exemplo 8: Informação nos atributos

```
<agenda>
  <entrada id="e1" tipo="pessoa" nome="José Carlos Ramalho"
           email="jcr@di.uminho.pt" telefone="253 604479"/>
  ...
</agenda>
```

Este documento é também um documento XML bem formado – o que mostra a fragilidade da fronteira entre atributo e elemento – mas, como veremos mais à frente, muito mais difícil de processar.

1.5.1 Atributos reservados

Há características universais que o conteúdo dos elementos pode partilhar em diferentes aplicações. Normalmente, incluem-se nestas características a língua utilizada e a importância dos caracteres brancos. Para evitar conflitos com nomes de atributos definidos pelo utilizador, a norma reservou o prefixo 'xml:'. Na norma XML há apenas dois atributos reservados, *'xml:lang' e 'xml:space'*, que se descrevem a seguir.

xml:lang – Este atributo pode ser associado a qualquer elemento e indica qual a língua em que o texto desse elemento está escrito. Esta faceta é útil em ambientes multilingues. Poderíamos ter por exemplo:

```
<para xml:lang="en">Hello</para>
<para xml:lang="pt">Olá</para>
<para xml:lang="fr">Bonjour</para>
```

Desta maneira poderíamos seleccionar o elemento a mostrar em dada altura ao utilizador, dependendo da escolha linguística deste. Os valores possíveis para este atributo estão definidos na norma ISO-639.

xml:space – Este atributo, também associável a qualquer elemento, pode ter um de dois valores: `default` ou `preserve`. Serve para indicar se o espaço branco no conteúdo do elemento em causa é ou não relevante. O valor `default` indica que o espaço não é importante e fará com que a maioria dos processadores compactem qualquer sequência de carácteres brancos num espaço. O valor `preserve` fará com que os caracteres brancos sejam mantidos como parte integrante do conteúdo do elemento.

1.6 Secções especiais de texto

Quando um autor quer usar caracteres que podem ser confundidos com aqueles utilizados para delimitar as anotações, como '<', '>' e '&', deve usar entidades como '<', '>' ou '&' no seu lugar. Apesar de uma entidade deste tipo ter o efeito desejado de eliminar a confusão, não é de todo intuitiva e o texto torna-se de difícil leitura. Se determinado pedaço de texto contiver muitos caracteres deste tipo, por exemplo um extracto de um documento XML que se queira usar como demonstração, a utilização de entidades pode ser considerada inaceitável.

Considere o seguinte exemplo:

Exemplo 9: Caracteres especiais no texto

Para obter o seguinte texto final:

```
Prima a tecla <<<ENTER>>>.
```

O utilizador deve introduzi-lo da seguinte forma:

```
Prima a tecla &lt;&lt;&lt;ENTER&gt;&gt;&gt;.
```

Para resolver as situações em que estes caracteres abundam, o XML dispõe de um mecanismo: *Secções Marcadas de Texto*. Numa secção destas, pode-se escrever livremente sem ser necessário substituir aqueles carácteres pelas entidades correspondentes.

Uma secção marcada de texto é sempre iniciada por `<![CDATA[` e terminada por `]]>`.

Exemplo 10: Secções marcadas de texto

Recorrendo ao mecanismo das *Secções Marcadas de Texto* para se obter o mesmo resultado final:

```
Prima a tecla <<<ENTER>>>.
```

o texto a introduzir terá a seguinte forma:

```
<![CDATA[Prima a tecla <<<ENTER>>>.]]>
```

Contudo, as referências a entidades, dentro destas secções, são ignoradas.

Exemplo 11: Secções marcadas e entidades

Assim, o resultado visual (final) do texto XML:

```
<![CDATA[Prima a tecla &lt;&lt;&lt;ENTER&gt;&gt;&gt;.]]>
```

será:

```
    Prima a tecla &lt;&lt;&lt;ENTER&gt;&gt;&gt;.
```

Terminou aqui a descrição dos elementos a que se pode recorrer para a constituição de um documento XML bem formado.

1.7 Regras de bem formação

Após termos visto quais os componentes de um documento XML, vamos enunciar um conjunto de regras que deve ser seguido na construção de um documento XML bem formado:

Um documento XML deve ter sempre uma declaração XML no início – Apesar de não ser obrigatória, muitos processadores não funcionarão correctamente se ela estiver ausente. Nos nossos documentos, em língua portuguesa, a declaração XML terá sempre a seguinte forma:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

O documento deve incluir um ou mais elementos – Para ser bem formado, um documento XML precisa de incluir um ou mais elementos. O primeiro, que delimitará todo o corpo do documento, é o elemento raiz e todos os outros deverão estar incluídos dentro dele.

Exemplo 12: Documento XML bem formado

No documento XML abaixo, o elemento raiz é `sumarios`, contendo, no seu interior, os elementos `disciplina`, `professor` e `aula`, os quais, por sua vez, são constituídos por outros elementos e texto:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<sumarios>
```

```
  <disciplina> Processamento Estruturado de
               Documentos</disciplina>
  <professor>
     <nome>José Carlos Ramalho</nome>
     <email>jcr@di.uminho.pt</email>
     <url>http://www.di.uminho.pt/~jcr</url>
  </professor>
  <aula tipo="T">
     <data>2000.10.02</data>
     <sumario>
        <p>
          Anotação de Documentos: um pouco de
          história.</p>
        <p>
          Linguagens de Anotação como meta-linguagens:
          o SGML e o XML.</p>
        <p>
          Anotação Descritiva. Ciclo de vida dos
          documentos estruturados.</p>
     </sumario>

  </aula>
  ...
</sumarios>
```

Todos os elementos têm anotações de início e fecho – A única excepção a esta regra são os elementos vazios, cujas anotações podem ser substituídas por uma única anotação de início que termina em '$/>$'.

Os elementos deverão estar aninhados correctamente – No exemplo seguinte, apresenta-se a situação a evitar.

Exemplo 13: Elementos aninhados incorrectamente

```
O título do livro é <B><I>XML</B> ... Prática</I>.
```

Ao contrário do que acontece no excerto acima, o primeiro elemento a começar terá sempre de ser o último a fechar e o último a abrir deverá ser sempre o primeiro a fechar. Neste caso, as anotações de `B` e de `I` estão cruzadas. O elemento `B` deveria terminar antes do `I` iniciar, ou iniciar depois do início de `I`, ou ainda fechar depois do fecho de `I`.

Os valores de atributos têm de estar dentro de aspas – Se numa dada situação o valor do atributo necessitar de conter aspas, então deverão ser usados apóstrofes (ou aspas simples) para limitar o valor, como se vê no exemplo abaixo:

```
<citação texto='O miúdo falou:"O Rei vai nú!"' />
```

1.8 *NameSpaces*

Usando XML, o utilizador tem uma liberdade total na definição da sua linguagem, podendo atribuir os nomes que bem entender aos seus elementos e atributos. No entanto, à medida que o número de utilizadores e de aplicações XML foi crescendo, surgiu um problema que não estava previsto; começaram a aparecer conflitos nos nomes dos elementos: quando num documento XML se importam pedaços de outros documentos XML escritos por outros autores, poderão surgir conflitos como anotações com o mesmo nome, mas com semânticas diferentes ou utilizadas em contextos contraditórios.

Exemplo 14: Conflitos entre elementos com o mesmo nome

Consideremos o caso do elemento `livro` usado em dois contextos diferentes: num catálogo e num pedido de encomenda.

Elemento livro no catálogo:

```
<livro>
  <titulo>XML: da teoria à prática</titulo>
  <resumo>Livro que cobre duma maneira geral mas com
    alguma profundidade a temática XML.</resumo>
  <temas>
    <tema>XML</tema>
    <tema>XSL</tema>
  </temas>
</livro>
```

Elemento livro no pedido de encomenda:

```
<encomenda>
  ... pagamento e envio ...
  <item>
    <livro>
      <titulo>XML: da teoria à prática</titulo>
      <isbn>1-999999-88-7</isbn>
    </livro>
    <quantidade>4</quantidade>
    <preço EUR="17 euros"/>
  </item>
</encomenda>
```

O elemento `livro` apresenta duas semânticas diferentes nos dois contextos em que é usado.

Se estivéssemos perante um outro documento formado com elementos de ambos os documentos apresentados, teríamos alguma dificuldade em interpretar os componentes `livro` que aparecessem, pois não saberíamos descobrir a proveniência desses elementos `livro`.

Os *NameSpaces* são a solução do problema.

Um *NameSpace* é uma superetiqueta formada pelo nome da anotação ao qual é concatenado um prefixo. Esse prefixo é definido pelo utilizador e deverá ser único.

Para garantir tal unicidade, convencionou-se que se usaria a sintaxe dos URL para o prefixo. Assim, partindo do princípio de que cada utilizador tem um URL próprio, acabam-se os conflitos.

1.8.1 Criação de *NameSpaces*

O *World Wide Web Consortium (W3C)* publicou uma recomendação com o título *Namespaces in XML*, onde se define um *Namespace* como:

> Uma colecção de nomes, identificados por uma referência URI (*Universal Resource Identifier*) que é usada nos documentos XML como prefixo dos nomes de elementos e de atributos.

A mesma recomendação define o modo como devem ser criados e declarados os *Namespaces*. Para esse efeito, convencionou-se existir um atributo *global* (pode ser instanciado em qualquer elemento do documento) de nome `xmlns`. É com este atributo que se define/cria um *NameSpace*.

No exemplo seguinte, apresentam-se algumas declarações de *NameSpaces* válidos:

Exemplo 15: *NameSpaces*

```
xmlns="http://xml.di.uminho.pt/XMLSamples/livro.dtd"
xmlns="livro.dtd"
xmlns="meu-URL/DTDs/livro.dtd"
```

As três declarações são válidas, apesar de só a primeira seguir a recomendação de utilizar o formato de um URI para o identificador do *NameSpace*.

1.8.2 Prefixos

A *string* identificadora de um *NameSpace* é demasiado grande para ser manipulada directamente. Assim, e para que a utilização de um *NameSpace* seja viável, associa-se-lhe uma abreviatura (um identificador ou qualificador do nome) que pode, mais tarde, ser utilizado como referência ao *NameSpace*. Este qualificador é também, muitas vezes, designado por prefixo.

Exemplo 16: *NameSpaces* com prefixos

Em relação ao último exemplo, os *NameSpaces* podiam ter sido declarados assim:

```
xmlns:catalogo="http://xml.di.uminho.pt/Samples/livro.dtd"
xmlns:encomenda="livro.dtd"
xmlns:jcr="meu-URL/DTDs/livro.dtd"
```

Neste caso, `catalogo`, `encomenda` e `jcr` seriam abreviaturas dos respectivos *NameSpaces* indicados à frente.

Imaginemos agora um cenário ligeiramente diferente: temos um catálogo de livros em XML ao qual queremos acrescentar os nossos comentários usando as nossas anotações para comentários. Podíamos utilizar os *NameSpaces* da seguinte forma:

Exemplo 17: Utilização de *NameSpaces* com prefixo

```
<catalogo:livro
  xmlns:catalogo="http://xml.di.uminho.pt/Samples/livro.dtd"
  xmlns:jcr="http://jcr.di.uminho.pt/DTD/coment.dtd">

  <catalogo:titulo>XML: da teoria à prática
  </catalogo:titulo>
  <jcr:opiniao>Um bom ponto de partida ...</jcr:opiniao>
```

```
  <catalogo:resumo>Livro que cobre duma maneira geral mas com
      alguma profundidade a temática XML.
  </catalogo:resumo>
  ...
</catalogo:livro>
```

1.8.3 *NameSpaces* locais

Como se disse atrás, o atributo `xmlns` pode ser utilizado em qualquer elemento e não apenas no elemento raiz. Assim, e para o tornar mais claro, o exemplo anterior poderia ser reescrito da seguinte maneira:

Exemplo 18: Utilização de NameSpaces locais

```
<catalogo:livro
    xmlns:catalogo="http://xml.di.uminho.pt/DTD/livro.dtd">
  <catalogo:titulo>XML: da teoria à prática
  </catalogo:titulo>
  <jcr:opiniao
      xmlns:jcr="http://jcr.di.uminho.pt/DTD/coment.dtd">
    Um bom ponto de partida ...</jcr:opiniao>
  <catalogo:resumo>Livro que cobre duma maneira geral mas
      com alguma profundidade a temática XML.
  </catalogo:resumo>
  ...
</catalogo:livro>
```

A declaração do *Namespace* ficou agora junto do elemento que faz uso dela (`jcr:opiniao`).

1.8.4 *NameSpaces* por omissão

O atributo `xmlns` pode ser utilizado isoladamente sem a declaração de um prefixo. Quando isso acontece, nenhum prefixo é especificado e diz-se que se acabou de declarar um *NameSpace* por omissão. Todos os elementos filho do elemento onde o *NameSpace* foi declarado pertencem agora a esse *NameSpace*.

O mesmo exemplo do catálogo com comentários pode então ser reescrito da seguinte forma:

Exemplo 19: Utilização de *NameSpaces* por omissão

```
<livro
    xmlns="http://xml.di.uminho.pt/DTD/livro.dtd">
  <titulo>XML: da teoria à prática</titulo>
  <jcr:opiniao
      xmlns:jcr="http://jcr.di.uminho.pt/DTD/coment.dtd">
    Um bom ponto de partida ...</jcr:opiniao>
  <resumo>Livro que cobre duma maneira geral mas com
      alguma profundidade a temática XML.
  </resumo>
  ...
</livro>
```

Como o *Namespace* associado a `livro` foi declarado sem prefixo, todos os elementos filhos de `livro` são considerados como pertencentes a esse *NameSpace*, à excepção daqueles que tenham um *NameSpace* declarado localmente (como é o caso de `jcr:opiniao`).

Mais à frente, veremos a relação dos *NameSpaces* com os DTD e com os XML Schema. Mais tarde, iremos utilizá-los quando estivermos a discutir o XSL.

Para terminar o capítulo, vamos ver uma variante do XML que está relacionada com a representação abstracta da informação num documento.

1.9 Forma canónica de documentos XML

Nos últimos tempos, tem surgido uma preocupação crescente no seio da comunidade de investigadores e utilizadores do XML: como representar, de uma forma abstracta, a informação num documento XML de modo a ser possível comparar documentos?

Além de ser uma questão actual, achamos pertinente colocá-la aqui porque introduz mais um conceito que aparece de vez em quando a quem trabalha com XML: o XML canónico.

A solução utilizada noutras áreas do conhecimento científico é a redução dos dois objectos que se querem comparar à forma mais simples, utilizando uma metodologia convencional. Noutras áreas, esta operação recebeu o nome de redução à forma canónica. Aqui, designaremos o resultado dessa redução por XML canónico.

O XML canónico é uma proposta existente no *World Wide Web Consortium (W3C)* e que o leitor mais interessado poderá consultar na *Web*[2].

Na representação em XML canónico, dois documentos XML logicamente equivalentes, mas que podiam estar representados de maneiras diferentes em XML (espaçamento, linhas em branco, codificação, elementos vazios, ...), são iguais *byte* a *byte*. Isto é conseguido à custa de uma sintaxe bastante estrita: só é permitida a codificação de caracteres em UTF-8, os pares CR/LF são substituídos por LF, nos elementos textuais as sequências de caracteres brancos são substituídas por um único espaço, ...

Há vários pacotes de *software* que integram ferramentas que fazem a redução de documentos XML à forma canónica. Um deles é o conjunto de ferramentas da IBM *Alphaworks*, o XML4J[3].

Concluindo, dois documentos XML na forma canónica podem ser comparados directamente e qualquer diferença entre eles será bem evidente, podendo este processo de comparação ser automatizado.

No próximo capítulo, vamos abordar a especificação de DTD e o conceito de **documento XML válido**.

[2] http://www.w3.org/TR/xml-c14n
[3] http://www.alphaworks.ibm.com/tech/xml4j

2 Documentos XML válidos

Até ao momento vimos como é relativamente fácil ao utilizador comum criar documentos XML bem formados. Um documento XML bem formado não necessita de a ele ter associado um processo de validação exaustivo, o que simplifica muito o seu processamento.

Em muitas aplicações XML, os documentos são gerados automaticamente. Nestas aplicações, o gerador garante por si só a bem-formação e a validação dos documentos e, nesta situação, trabalhar com documentos bem formados é mais do que suficiente. Porém, na maioria dos casos em que é o utilizador que edita e manipula os documentos XML, há necessidade de estipular um conjunto de regras que estabeleçam a validade dos documentos. Este conjunto de regras define uma classe, ou tipo, de documento e permite a posterior validação e processamento dos documentos criados.

À especificação de um tipo de documentos dá-se o nome de DTD (*Document Type Definition*) ou XML Schema. O processo que verifica se um documento está de acordo com um DTD ou Schema designa-se por **Validação**; a um documento depois de verificado atribuímos a classificação de **documento XML Válido**.

Os XML Schemas serão discutidos num capítulo mais à frente (cap. **??**). Neste capítulo, apresenta-se em detalhe a linguagem com a qual se especifica um DTD.

2.1 Componentes de um documento XML válido

Um documento XML válido é composto por duas grandes e distintas partes: o DTD e a Instância. O DTD define as regras a que a instância tem de obedecer para pertencer ao tipo de documento definido pelo DTD.

Como já foi abordado no capítulo anterior, a anotação de um elemento é composta por duas marcas, uma de início do elemento e uma de fim do elemento. Estas anotações irão descrever as qualidades características do elemento. Uma daquelas características é o seu nome, que identifica o tipo do elemento: `parágrafo`, `figura`, `lista`, ...

Adicionalmente, um elemento poderá ser qualificado por mais características que lhe são inerentes. Estas recebem a designação de atributos.

As anotações descrevem a estrutura de um documento. Indicam quais os elementos que ocorrem no documento e em que ordem. Esta estrutura tem de ser válida de acordo com o conjunto de declarações no DTD que define todas as estruturas permitidas num determinado tipo de documento.

Para introduzir este conceito de DTD, recorre-se ao exemplo seguinte.

Exemplo 20: DTD para uma carta

Neste exemplo, todas as linhas começadas por `<!-` são comentários e destinam-se a esclarecer o objectivo geral do tipo de documento e os de cada elemento em particular.

```
<!-- Este DTD define a estrutura de docs do tipo CARTA -->
<!DOCTYPE CARTA [
<!-- Uma carta é uma sequência de elementos:           -->
<!--   um destinatário, um texto de abertura, um corpo -->
<!--   e um texto de fecho.                            -->
<!ELEMENT CARTA (DEST, ABERTURA, CORPO, FECHO)>
<!-- O destinatário é texto.                           -->
<!ELEMENT DEST (#PCDATA)>
<!-- A abertura é texto.                               -->
<!ELEMENT ABERTURA (#PCDATA)>
```

```
<!-- O corpo é composto por um ou mais parágrafos.     -->
<!ELEMENT CORPO (PARA)+>
<!-- Um parágrafo é composto por texto podendo ter uma -->
<!-- ou mais listas intercaladas.                       -->
<!ELEMENT PARA (#PCDATA | LISTA)*>
<!-- Uma lista é uma sequência de um ou mais items.    -->
<!ELEMENT LISTA (LITEM)+>
<!-- Um item é texto.                                   -->
<!ELEMENT LITEM (#PCDATA)>
<!-- O fecho é texto.                                   -->
<!ELEMENT FECHO (#PCDATA)>
```

Este DTD é muito simples, contém apenas declarações de elementos. Mesmo assim, permite escrever *cartas* estruturadas e ainda fazer a validação estrutural desses documentos. Contudo, está ainda um pouco distante de um DTD real que contém normalmente maior riqueza informativa.

Formalmente, pode-se dizer que um DTD é composto por um conjunto de declarações. Existem quatro tipos de declarações que podem ser usadas num DTD: elementos, atributos, entidades e instruções de processamento.

As entidades estão relacionadas com a organização física de um documento XML, por isso iremos abordá-las no próximo capítulo (cap. **??**). As instruções de processamento já foram descritas no capítulo anterior (cap. 1). Nas próximas secções vamos analisar os outros tipos de declarações – elementos e atributos – visto que, esses sim, estão relacionados com a organização lógica de um documento, que é a questão mais importante a discutir neste capítulo.

2.2 Elementos

Um elemento é definido, no DTD, numa declaração do tipo `ELEMENT`, que obedece à seguinte estrutura:

```
<!ELEMENT  identificador (expressão-conteúdo)>
```

Em que:

identificador é o nome do elemento, o qual é depois usado nas anotações:

```
DTD:
  <!ELEMENT  meu-elem (...)>

Instância:
  ...<meu-elem>conteúdo do elemento</meu-elem>...
```

O *identificador* do elemento pode ser formado por letras, dígitos, dois pontos, hífens, *underscores* e pontos. Mas só pode começar por letra, *underscore* ou dois pontos.

expressão-conteúdo é a definição do conteúdo do elemento escrita numa linguagem de expressões regulares que obedece a uma álgebra ([**?**]). A expressão regular que define o conteúdo do elemento especifica que subelementos podem aparecer, em que ordem e em que número:

```
DTD:
  <!-- Um contacto é composto por uma sequência
       de: nome, email e telefone. -->
  <!ELEMENT  contacto (nome,email,telefone)>

Instância:
  ...
  <contacto>
    <nome>conteúdo de nome</nome>
    <email>conteúdo de email</email>
    <telefone>conteúdo de telefone</telefone>
  </contacto>
  ...
```

Relativamente ao conteúdo, podemos classificar os elementos como pertencendo a um de cinco tipos possíveis:

vazio – o elemento não deverá ter qualquer conteúdo; este tipo de elementos tem apenas um valor posicional.

textual – o conteúdo do elemento é apenas texto (com algumas restrições).

estruturado – o conteúdo do elemento é formado por uma combinação de outros elementos.

misto – o conteúdo do elemento é texto no meio do qual poderão surgir alguns elementos *soltos*.

livre – o conteúdo do elemento não tem qualquer restrição, pode ser uma mistura de texto com uma combinação qualquer de outros elementos definidos no DTD; este tipo de elementos será usado apenas nos estados iniciais do desenvolvimento de uma aplicação.

Vamos então ver como se pode definir o conteúdo de um elemento, recorrendo à álgebra de expressões regulares.

2.2.1 Álgebra do conteúdo

Esta álgebra não é mais do que um conjunto de definições com as quais podemos exprimir e definir a constituição de elementos num documento XML.

Assim, vamos ter constantes para conteúdos atómicos, e um conjunto de combinadores e operadores para a definição de conteúdos estruturados e semi-estruturados (que designaremos globalmente por conteúdos compostos).

Conteúdos atómicos

Elementos vazios: a constante `EMPTY`

Alguns elementos não têm conteúdo textual nem elementos filho. Designamos estes elementos por elementos vazios e usamos a palavra-chave `EMPTY` para definir o seu conteúdo.

Por exemplo, suponhamos que estamos a criar um documento XML e, num dado ponto, queremos inserir uma imagem. Como não se pode inserir a imagem no meio do documento (a imagem não está em XML – estamos a excluir as imagens SVG[1]),

[1] Standard Vector Graphics- linguagem XML para a definição de imagens vectoriais de pequena e média resolução.

coloca-se um elemento marcando o local onde se quer inseri-la, com um apontador para o ficheiro externo onde aquela se encontra armazenada. Tal elemento seria definido da seguinte forma:

```
<!ELEMENT imagem EMPTY>
```

Esta declaração faz saber ao processador de XML que `imagem` é um elemento vazio e que, por isso, será representado por uma única anotação:

```
...
O trajecto realizado encontra-se no mapa abaixo:
<imagem path="../imagens/mapa.jpg"/>
...
```

Elementos textuais: a constante #PCDATA

O conteúdo de elementos que são formados apenas por texto define-se recorrendo à palavra-chave #PCDATA (*Parsed Character Data*) que, como a expansão em inglês indica, é analisado pelo processador do documento. Esta análise é necessária para expandir as referências a entidades que possam existir nesse texto. Há, no entanto, algumas restrições aos caracteres que se podem usar:

- o carácter & não pode ser utilizado; em seu lugar deverá ser colocada uma referência à entidade que o define - `&`
- o carácter $<$ não pode ser utilizado; em seu lugar deverá ser colocada uma referência à entidade que o define - `<`
- o carácter $>$ não pode ser utilizado; em seu lugar deverá ser colocada uma referência à entidade que o define - `>`

O processador de XML encarrega-se, como já foi referido, de substituir as entidades pelos caracteres correspondentes.

Exemplo 21: Caracteres reservados

Suponha que, na pele de um professor, redigia um texto em XML sobre a avaliação de uma disciplina e, em determinada altura, necessitava de escrever:

```
...se a nota for > 8 e < 10 o aluno deve ser
submetido a uma prova oral.
```

Para obedecer às restrições impostas pelo XML o texto deveria ser escrito da seguinte forma:

```
...se a nota for &gt; 8 e &lt; 10 o aluno deve ser
submetido a uma prova oral.
```

Tipicamente, um documento contém texto e embora, como veremos mais à frente, possam surgir documentos XML especiais só com anotações, esta é a situação normal. Por isso, verifica-se que elementos com conteúdo #PCDATA existem em praticamente todos os documentos XML. Por exemplo, para organizar simplesmente um dado texto em parágrafos, bastaria definir o seguinte elemento:

```
<!ELEMENT para (#PCDATA)>
```

e então usá-lo ao longo do dito texto para marcar o início e o fim de cada parágrafo, obtendo-se um documento estruturado com o seguinte aspecto:

```
...
<para>Este livro tenta dar uma visão...</para>
<para>No primeiro capítulo, ...</para>
...
```

Note que #PCDATA é o único tipo de dados primitivo possível, quando se usa um DTD para especificar a estrutura de um documento XML. Mesmo que haja elementos cujo conteúdo seja numérico ou temporal, eles terão de ser especificados como tendo conteúdo textual. Esta é uma das limitações de um DTD que os XML Schema resolvem, como iremos ver mais à frente (cap. **??**).

Conteúdos compostos

Elementos estruturados

É neste tipo de elementos que surge a álgebra para especificação de conteúdos. Um elemento deste tipo define-se como uma combinação de outros elementos. Para definir estas combinações, temos dois conjuntos de operadores: *operadores de conexão* e *operadores de ocorrência*.

Operadores de conexão:

Os operadores de conexão são normalmente colocados entre dois elementos e definem a ordem em que estes podem ocorrer ou combinar-se.

, operador de sequência (a, b) — significa que o elemento tem de ser composto por um elemento a e um elemento b, e que a deve preceder b.

No exemplo 20, de uma carta, aparece logo no início:

```
<!ELEMENT CARTA (DEST, ABERTURA, CORPO, FECHO)>
```

O que significa que uma carta é, obrigatoriamente, constituída por um destinatário, uma abertura, um corpo e um fecho, nesta ordem. A ordem sequencial é imposta pelo operador **','**.

| **operador de alternativa** $(a|b)$ — significa que o elemento é composto por um elemento a ou por um elemento b.

Nesse mesmo exemplo 20, a declaração:

```
<!ELEMENT PARA (#PCDATA | LISTA)*>
```

significa que o conteúdo de PARA pode ser formado, alternativamente, por texto ou por subelementos LISTA.

Operadores de ocorrência:

Os operadores de ocorrência são aplicados a um termo (um elemento ou uma expressão de conteúdo, i. e., elementos ligados por operadores) e visam limitar o número de ocorrências do termo ao qual são aplicados.

? (0 ou 1 vez) $a?$ — o elemento definido por esta expressão tem de ser constituído por um elemento a, que é opcional.

*** (0 ou mais vezes)** $a*$ — o elemento definido por esta expressão tem de ser constituído por zero ou mais elementos a.

+ (1 ou mais vezes) $a+$ — o elemento definido por esta expressão tem de ser constituído por um ou mais elementos a.

Elementos mistos

Um elemento de conteúdo misto pode definir-se como um elemento textual, onde podem ocorrer livremente, entre o texto, alguns elementos previamente identificados.

Num DTD, um elemento de conteúdo misto deve ser sempre definido como uma *alternativa* à qual é aplicado o *operador de ocorrência* *. Os operandos da alternativa são `#PCDATA` e os elementos que podem ocorrer livremente no texto. A declaração de um elemento deste tipo tem a seguinte forma:

```
<!ELEMENT elem-misto (#PCDATA | elem1 | elem2 | ...)*>
```

Os elementos com conteúdo misto, ou semi-estruturados, são aqueles que mais problemas levantam a quem constrói aplicações baseadas em XML. No entanto, há situações em que eles são mesmo necessários e não é possível evitá-los: normalmente em aplicações de publicação electrónica onde o conteúdo dos documentos é semi-estruturado.

Um dos exemplos mais comuns é o elemento parágrafo: constituído por texto no meio do qual podem surgir livremente datas, nomes, ...

```
<!ELEMENT paragrafo (#PCDATA | data | nome)*>
```

Uma instância deste elemento poderia ser:

```
... <paragrafo>E foi então, <data>1150</data>, que
D. <nome>Afonso Henriques</nome> se
instalou ... </paragrafo> ...
```

Elementos livres: a constante ANY

Os elementos deste tipo acabam por ser elementos de conteúdo misto onde, no meio do texto, pode aparecer livremente qualquer um dos outros elementos definidos no DTD, um número de vezes também livre. O conteúdo destes elementos é definido pela palavra-chave ANY.

Exemplo 22: Elementos livres: a constante ANY

Considere o seguinte DTD:

```
<!ELEMENT doc (para)+>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT lugar (#PCDATA)>
<!ELEMENT data (#PCDATA)>
<!ELEMENT para ANY>
```

Neste DTD, define-se um tipo de documentos com elemento raiz `doc`, composto por um ou mais elementos `para`. Por sua vez, o elemento `para` está definido como sendo de conteúdo ANY, ou seja, pode ser texto com elementos `nome`, `lugar` ou `data` pelo meio.

De acordo com este DTD, o documento abaixo seria um documento XML válido.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<doc>
  <para>Este é o primeiro paragráfo.</para>
```

```
  <para>Este é um parágrafo com nomes. A <nome>
  Ana</nome> comeu a papa da <nome>Joana</nome>.</para>
  <para>Tudo aconteceu em <lugar>Braga</lugar>...</para>
  <para>O parágrafo anterior contém lugares.</para>
</doc>
```

2.2.2 Exemplos: o DTD *Agenda* e o DTD *Poema*

Vamos, agora, aplicar, em dois exemplos, o que se acabou de descrever.

Como já foi subtilmente referido, o XML serve para modelar informação rigidamente estruturada, semelhante àquela que se guarda numa base de dados, ou, também, para modelar informação semi-estruturada onde os conteúdos mistos abundam. Os próximos exemplos visam ilustrar estas duas situações.

Como exemplo da primeira superclasse, informação estruturada, apresentamos a agenda de contactos. E, como exemplo do segundo caso, um poema.

Exemplo 23: A agenda

No capítulo 1, usou-se uma agenda de contactos para exemplificar alguns dos componentes de um documento XML. Agora, vai-se definir num DTD a estrutura de uma agenda.

Uma agenda deste tipo é, essencialmente, constituída por uma lista de entradas. Uma entrada pode ser simples, correspondendo a uma pessoa ou empresa, ou pode ser composta e, neste caso, corresponde a um grupo de pessoas ou empresas. De notar que num grupo podem, também, aparecer referências a entradas ou grupos previamente definidos na agenda. A agenda poderá conter um número livre de grupos aninhados.

Cada entrada é constituída pelos seguintes itens de informação:

ident – uma string que corresponderá ao identificador único de uma entrada ou grupo.

tipo – campo com dois valores possíveis: pessoa ou empresa.

nome – nome da pessoa ou da empresa.

email – *e-mail* da pessoa ou empresa.

telefone – telefone da pessoa ou empresa.

ref – uma *string* contendo o identificador de uma entrada ou de um grupo previamente definido.

Um exemplo possível de um documento XML contendo uma agenda apresenta-se a seguir:

```
<AGENDA>
  <ENTRADA>
    <IDENT>e1</IDENT>
    <TIPO>pessoa</TIPO>
    <NOME>José Carlos Ramalho</NOME>
    <EMAIL>jcr@di.uminho.pt</EMAIL>
    <TELEFONE>253 604479</TELEFONE>
  </ENTRADA>
  <GRUPO>
    <IDENT>epl</IDENT>
    <ENTRADA>
      <IDENT>e2</IDENT>
      <TIPO>pessoa</TIPO>
      <NOME>Pedro Henriques</NOME>
      <EMAIL>prh@di.uminho.pt</EMAIL>
      <TELEFONE>253 604469</TELEFONE>
    </ENTRADA>
    <ENTRADA>
      <IDENT>e3</IDENT>
      <TIPO>pessoa</TIPO>
      <NOME>João Saraiva</NOME>
      <EMAIL>jas@di.uminho.pt</EMAIL>
      <TELEFONE>253 604479</TELEFONE>
    </ENTRADA>
    <REF>e1</REF>
  </GRUPO>
  <ENTRADA>
```

```
        <IDENT>e4</IDENT>
        <TIPO>pessoa</TIPO>
        <NOME>José João Almeida</NOME>
        <EMAIL>jj@di.uminho.pt</EMAIL>
        <TELEFONE>253 604433</TELEFONE>
      </ENTRADA>
    </AGENDA>
```

Um DTD para este tipo de documentos podia ser especificado com o conjunto de declarações seguinte:

```
<!ELEMENT AGENDA (ENTRADA | GRUPO)+>
<!ELEMENT ENTRADA (IDENT, TIPO, NOME, EMAIL, TELEFONE)>
<!ELEMENT GRUPO (IDENT,(ENTRADA | GRUPO | REF)+)>
<!ELEMENT NOME (#PCDATA)>
<!ELEMENT EMAIL (#PCDATA)>
<!ELEMENT TELEFONE (#PCDATA)>
<!ELEMENT IDENT (#PCDATA)>
<!ELEMENT TIPO (#PCDATA)>
<!ELEMENT REF (#PCDATA)>
```

Observe no DTD acima o uso dos operadores de conexão `,` e $|$ e do operador de ocorrência $+$, bem como o constructor constante `#PCDATA`.

Note que a definição do elemento `grupo` é recursiva, isto é, um dos elementos do seu conteúdo pode ser novamente `grupo`. Em XML é possível definir estruturas lógicas recursivas.

No exemplo anterior, como já era previsto, não surgiu nenhum elemento de conteúdo misto. No próximo exemplo, a ideia é pegar num documento tradicional (semi-estruturado) e especificá-lo recorrendo a elementos de conteúdo misto.

Exemplo 24: O poema

Neste exemplo, vamos definir um DTD para um poema, mais especificamente para um soneto, da autoria de Álvaro de Campos. Em termos de anotação, será nossa intenção explicitar a estrutura (título, autor, quadras, tercetos e versos), bem como marcar nomes de pessoas ou lugares e datas que possam surgir no conteúdo textual.

A instância devidamente anotada tem o seguinte aspecto:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<poema>
  <titulo>"Soneto Já Antigo"</titulo>
  <autor>(Álvaro de Campos)</autor>
  <corpo>
    <quadra>
      <verso>Olha, <nome>Daisy</nome>: quando eu morrer
             tu hás-de</verso>
      <verso>dizer aos meus amigos aí de
             <lugar>Londres</lugar>,</verso>
      <verso>embora não o sintas, que tu escondes</verso>
      <verso>a grande dor da minha morte. Irás de</verso>
    </quadra>
    <quadra>
      <verso><lugar>Londres</lugar> p'ra <lugar>Iorque
             </lugar>, onde nasceste (dizes</verso>
      <verso>que eu nada que tu digas acredito),</verso>
      <verso>contar áquele pobre rapazito</verso>
      <verso>que me deu horas tão felizes,</verso>
    </quadra>
    <terceto>
      <verso>embora não o saibas, que morri...</verso>
      <verso>Mesmo ele, a quem eu tanto julguei
             amar,</verso>
      <verso>nada se importará... Depois vai dar </verso>
    </terceto>
    <terceto>
      <verso>a notícia a essa estranha
             <nome>Cecily</nome></verso>
      <verso>que acreditava que eu seria grande...</verso>
```

```
      <verso>Raios partam a vida e quem lá ande!</verso>
    </terceto>
  </corpo>
  <data>(1922)</data>
</poema>
```

O DTD, que permitirá validar esta instância, pode ser concebido de forma mais específica para esta família de poemas ou, em termos mais genéricos, para suportar qualquer tipo de poema. Neste caso, optámos por torná-lo mais específico e, então, definimos um DTD para sonetos.

```
<!ELEMENT poema   (titulo, autor, corpo, data) >
<!ELEMENT titulo  (#PCDATA)>
<!ELEMENT autor   (#PCDATA)>
<!ELEMENT corpo   (quadra,quadra,terceto,terceto)>
<!ELEMENT quadra  (verso, verso, verso, verso)>
<!ELEMENT terceto (verso, verso, verso)>
<!ELEMENT verso   (#PCDATA|nome|lugar)*>
<!ELEMENT nome    (#PCDATA)>
<!ELEMENT lugar   (#PCDATA)>
<!ELEMENT data    (#PCDATA)>
```

Como foi dito, note-se que temos agora um elemento com conteúdo misto: `verso`. Este elemento, que corresponde ao conteúdo semi-estruturado do poema, está definido como uma mistura de texto com elementos `nome` e `lugar`.

Se quiséssemos tornar este DTD mais genérico, podíamos redefinir o elemento `corpo` da seguinte forma:

```
<!ELEMENT corpo   (quadra|terceto)+>
```

Assim, o corpo do poema deixaria de ter de ser formado por duas quadras e dois tercetos e poderia ser formado por uma sequência de quadras e tercetos ordenados livremente.

Podíamos ainda torná-lo mais genérico, fazendo as seguintes redefinições:

```
<!ELEMENT corpo   (estrofe)+>
<!ELEMENT estrofe (verso)+>
```

Agora, um poema é formado por uma sequência de estrofes que, por sua vez, são formadas por uma sequência de um ou mais versos. Este DTD deverá dar para anotar qualquer tipo de poema.

Ao longo desta generalização, ganhou-se universalidade, mas perdeu-se riqueza semântica. A primeira versão garantia que um documento, que fosse válido de acordo com esse DTD, teria a estrutura de um soneto. A última versão ainda garante que um soneto devidamente anotado é um documento válido, mas, além disso, garante o mesmo para muitos outros poemas que não são sonetos.

Fica aqui, então, o aviso de que a generalização pode ser perigosa pois implica perdas semânticas. O ideal é termos o objectivo final bem claro: neste caso se quiséssemos processar sonetos – por exemplo, verificar métricas e rimas, das quadras e tercetos, optaríamos pela primeira versão do DTD; se, por outro lado, quiséssemos processar poemas num sentido mais lato, optaríamos pela última versão.

2.3 Atributos

No capítulo anterior (capítulo 1), abordou-se o conceito de atributo e a sua relação com os elementos. Agora, vamos ver como é que aqueles se podem declarar num DTD.

Um dos exemplos da secção anterior foi a agenda de contactos (ex. 23). No DTD apresentado estão presentes as seguintes declarações:

```
<!ELEMENT IDENT (#PCDATA)>
<!ELEMENT TIPO (#PCDATA)>
<!ELEMENT REF (#PCDATA)>
```

Na altura, foram todos definidos como elementos porque estávamos a utilizar apenas elementos na especificação do DTD. Agora, introduzindo os atributos, podem-se questionar algumas das decisões tomadas. Por exemplo, pensando bem, `TIPO` é uma característica do elemento `ENTRADA` e não um dos elementos do seu conteúdo; por isso pode passar a atributo (uma decisão sempre subjectiva por mais argumentos que se arranjem). Por seu lado, `IDENT` e `REF` devem ser vistos como um par pois é

com os seus valores que se monta um mecanismo de referenciação[2]. Ora, para este fim, existe no XML, um par de tipos de atributos, `ID` e `IDREF` que têm a eles associada a validação que garante que a referenciação está bem feita. Por este motivo, os elementos `IDENT` e `REF` deveriam, também, ser convertidos em atributos.

Assim, substituímos aquelas declarações iniciais pelas que se seguem:

```
<!ATTLIST ENTRADA IDENT ID #REQUIRED>
<!ATTLIST GRUPO IDENT ID #REQUIRED>
<!ATTLIST REFERENCIA REF IDREF #REQUIRED>
```

A primeira declaração associa um atributo de nome `IDENT` ao elemento `ENTRADA`, a segunda associa um atributo de nome `IDENT` ao elemento `GRUPO` e a terceira, associa um atributo de nome `REF` ao elemento `REFERENCIA`.

Note que, neste último, foi necessário criar um elemento, `REFERENCIA`, ao qual pudesse ser associado o atributo `REF`. Uma vez que este novo elemento não terá qualquer outro conteúdo, seja ele texto ou elementos filho, é declarado como um elemento vazio:

```
<!ELEMENT REFERENCIA EMPTY>
```

Depois destas alterações, o DTD da agenda ficou com o seguinte conjunto de declarações:

```
<!ELEMENT AGENDA (ENTRADA | GRUPO)+>
<!ELEMENT ENTRADA (NOME, EMAIL, TELEFONE)>
<!ELEMENT GRUPO (ENTRADA | GRUPO | REFERENCIA)+>
<!ELEMENT REFERENCIA EMPTY>
<!ELEMENT NOME (#PCDATA)>
<!ELEMENT EMAIL (#PCDATA)>
<!ELEMENT TELEFONE (#PCDATA)>

<!ATTLIST ENTRADA
            IDENT ID #REQUIRED
            TIPO  (pessoa|empresa) "pessoa">
```

[2]Mecanismo através do qual podemos referir, ou chamar, num dado ponto do documento, um elemento definido noutro ponto, identificando-o sem ambiguidades, ou seja, univocamente.

```
<!ATTLIST GRUPO IDENT ID #REQUIRED>
<!ATTLIST REFERENCIA REF IDREF #REQUIRED>
```

Apesar de ser uma decisão algo subjectiva, como se disse anteriormente, obtém-se uma definição do tipo de documento pretendido, um DTD, que é muito mais preciso e conciso em termos do papel desempenhado por cada componente do sistema de anotação.

2.3.1 Declaração

A declaração de atributos num DTD tem a seguinte sintaxe:

```
<!ATTLIST  elem-id
             att1-id  att1-tipo  att1-class att1-val-omissao
             ...
             attn-id  attn-tipo  attn-class attn-val-omissao
>
```

Onde:

elem-id é o identificador do elemento ao qual os atributos ficarão associados.

atti-id é o identificador do atributo `i`.

atti-tipo é o tipo do atributo `i`.

atti-class é a indicação da classe a que o atributo `i` pertence.

atti-val-omissão é a indicação do valor por omissão do atributo `i`.

Como se pode ver pela estrutura acima, uma declaração `ATTLIST` pode servir para declarar todos os atributos de um elemento. Usar uma declaração deste tipo para todos ou declarar cada um por sua vez, só depende do gosto pessoal de quem escreve o DTD. Para exemplificar este ponto, apresenta-se a seguir um conjunto de declarações para um elemento `imagem`.

Exemplo 25: Declarações de atributos

Se quiséssemos especificar um elemento `imagem` com atributos `ficheiro`, `largura` e `altura`, podíamos fazê-lo da maneira seguinte:

```
<!ELEMENT imagem EMPTY>

<!ATTLIST imagem
            ficheiro CDATA #REQUIRED
            largura  CDATA #IMPLIED
            altura   CDATA #IMPLIED
>
```

ou, alternativamente:

```
<!ELEMENT imagem EMPTY>

<!ATTLIST imagem ficheiro CDATA #REQUIRED>
<!ATTLIST imagem largura  CDATA #IMPLIED>
<!ATTLIST imagem altura   CDATA #IMPLIED>
```

Na primeira versão, os atributos são especificados numa única declaração; na segunda, cada atributo é especificado numa declaração. Ambas as versões são válidas em XML, só dependem do gosto do analista.

2.3.2 Tipos

Os valores dos atributos podem ser mais do que simples *strings*, podem ter uma semântica mais específica. Por exemplo, um atributo pode ser declarado de modo a que o valor que a ele fique associado seja único, i. e., um *identificador chave* que pode ser usado para referenciar e localizar um elemento num documento.

Em XML, existem dez tipos de atributo, que serão explicados a seguir:

- CDATA

- Enumerado
- ID
- IDREF
- IDREFS
- ENTITY
- ENTITIES
- NOTATION
- NMTOKEN
- NMTOKENS

Apenas estes tipos são permitidos. Em XML, não há maneira de especificar que o valor de um atributo é, por exemplo, do tipo inteiro ou data. Isto é, não há tipos definidos pelo utilizador, nem tipos compostos.

CDATA

Este é o tipo de atributo mais geral. O valor dum atributo deste tipo é qualquer texto livre sem anotações. Pode ser usado para muitas espécies diferentes de dados[3]: preços, URLs, endereços de *e-mail*, e todas aquelas que não se enquadrarem num dos outros tipos.

Os atributos do elemento `imagem`, no exemplo acima, são deste tipo.

ID

Um atributo do tipo ID deve conter um identificador (a obedecer às mesmas regras que o nome de um elemento ou atributo) que é único em todo o documento. Por outras palavras, no documento, não pode haver mais nenhum atributo deste tipo

[3] Como iremos ver (cap. **??**), os XML Schemas permitem definir tipos de dados mais apropriados para alguns destes casos.

com o mesmo valor. Por sua vez, cada um dos elementos não pode ter mais do que um atributo deste tipo.

Como a própria palavra-chave sugere (`ID`), este tipo de atributos deve ser usado para associar identificadores únicos a elementos.

Apesar de ser comum, basta olhar para o exemplo da agenda (ex. 23), os nomes dos atributos deste tipo não necessitam de ser obrigatoriamente chamados `ID` ou `id`, podendo ter nomes como: `ident`, `entrada-i` ou `código`.

No caso da agenda, há dois atributos deste tipo. Um associa um identificador único a cada `entrada`. O outro associa um identificador único a cada `grupo`.

```
<!ATTLIST ENTRADA
            IDENT ID #REQUIRED
            ... >
<!ATTLIST GRUPO IDENT ID #REQUIRED>
```

IDREF

Um atributo do tipo `IDREF` *aponta* para um atributo do tipo `ID` de outro elemento no documento. Este tipo de atributos é, normalmente, usado para implementar relações entre elementos, em que cada relação é representada por um par de atributos `ID/IDREF`.

As relações podem ser simples – de um para um ou de um para muitos, ou complexas – de muitos para muitos. Como exemplo das primeiras, existe o problema típico das referências bibliográficas: há um conjunto de registos bibliográficos, em que cada registo tem a ele associado um atributo do tipo ID que o identifica; sempre que se quer fazer referência a um registo usa-se um elemento vazio com um atributo do tipo `IDREF`; desta maneira a relação é estabelecida por um par `ID/IDREF`. No caso da agenda de contactos temos uma situação semelhante com as entradas e as referências:

```
<!ATTLIST ENTRADA IDENT ID #REQUIRED>
<!ATTLIST REFERENCIA REF IDREF #REQUIRED>
```

Uma instância XML possível seria:

```
...
<AGENDA>
  <ENTRADA IDENT="e1" TIPO="pessoa">
        ...
  </ENTRADA>
  <GRUPO IDENT="epl">
    <ENTRADA IDENT="e2" TIPO="pessoa">
          ...
    </ENTRADA>
    <ENTRADA IDENT="e3" TIPO="pessoa">
          ...
    </ENTRADA>
    <REFERENCIA REF="e1"/>
    <REFERENCIA REF="e5"/>
  </GRUPO>
     ...
</AGENDA>
```

Nesta instância, são utilizados dois elementos `REFERENCIA` que estabelecem relações com elementos `ENTRADA` ou `GRUPO` usando um par de atributos `ID/IDREF`. A primeira `REFERENCIA` está correcta pois usa o valor `e1` que existe como valor de um atributo `IDENT` associado a uma `ENTRADA`. A segunda, no entanto, está incorrecta pois o valor usado, `e5`, não existe na instância. Esta situação irá provocar um erro na validação do documento.

As relações de muitos para muitos representam outra situação onde é necessário utilizar este tipo de atributos. Por exemplo: estamos a registar, no mesmo documento, informação sobre livros e informação sobre autores; um livro pode ter sido escrito por vários autores e um autor pode ter escrito vários livros. Há duas soluções para representar esta informação: com repetição de informação, colocando dentro da informação de cada livro a informação dos seus autores; ou sem repetição de informação, construindo dois blocos disjuntos de informação, autores e livros, e utilizando pares de atributos `ID/IDREF` para implementar a relação.

Para além do tipo `CDATA`, os pares `ID/IDREF` representam o tipo de atributos mais utilizado em documentos XML associados a um DTD.

IDREFS

O tipo `IDREFS` é uma característica útil quando se pretende fazer referência a mais do que um elemento. O valor do atributo deve ser constituído por uma lista de identificadores separados por espaço, identificadores esses que devem estar instanciados num atributo do tipo `ID` algures no documento.

Considere o exemplo anterior dos livros e respectivos autores:

```
<!ELEMENT AUTOR ...>
<!ATTLIST AUTOR
            id ID #REQUIRED>
...
<!ELEMENT LIVRO ...>
<!ATTLIST LIVRO autores IDREFS #REQUIRED>
```

Se quiséssemos representar a informação referente ao presente livro, teríamos:

```
...
<AUTOR id="jcr">...</AUTOR>
<AUTOR id="prh">...</AUTOR>
...
<LIVRO autores="jcr prh"> ... </LIVRO>
...
```

Enumerado

O tipo enumerado não utiliza uma palavra-chave como os outros tipos de atributo; em vez disso, oferece uma lista (ou enumeração) de valores possíveis para esse atributo. Cada um dos valores possíveis tem de ser *um identificador XML* (tem que seguir as mesmas regras que os identificadores de elementos).

Voltando ao caso de estudo da agenda, imagine que agora queremos distinguir as entradas que dizem respeito a pessoas, empresas (entidades privadas) e instituições (entidades públicas). Vamos fazê-lo recorrendo a um atributo do tipo `enumerado` de nome `tipo` com valores possíveis `pessoa`, `empresa`, e `instituicao`:

```
<!ELEMENT ENTRADA (NOME, EMAIL, TELEFONE)>
```

```
<!ATTLIST ENTRADA
            IDENT ID #REQUIRED
            TIPO (pessoa|empresa|instituicao) "pessoa">
```

Como se pode ver, a lista de valores possíveis define-se dentro de parêntesis curvos com uma barra vertical a separar os valores. O valor por omissão, colocado entre aspas a seguir à enumeração, deverá ser um dos valores da lista e corresponde ao valor que será assumido pelo processador caso o utilizador não atribua um valor ao atributo.

Poderíamos, então, ter a seguinte instância XML:

```
...
<ENTRADA IDENT="e9" TIPO="empresa">
  <NOME>Lavandaria Raio de Lua</nome>
  <EMAIL>raio@mail.pt</EMAIL>
  <TELEFONE>253 610444</TELEFONE>
</ENTRADA>
...
```

ENTITY

Um atributo do tipo `ENTITY` pode conter como valor o nome de uma entidade (cap. **??**) declarada algures no DTD.

Por exemplo, o elemento `ENTRADA` podia ter uma fotografia (em formato JPEG ou outro), associada através de um atributo `foto`:

```
<!ELEMENT ENTRADA (NOME, EMAIL, TELEFONE)>

<!ATTLIST ENTRADA
            IDENT ID #REQUIRED
            TIPO (pessoa|empresa|instituicao) "pessoa"
            FOTO ENTITY >
```

Se no DTD existisse uma entidade de nome `paulo` com a fotografia do `Paulo José`, podia usar-se este atributo para associar a entidade à entrada:

```
...
<ENTRADA IDENT="e7" FOTO="paulo">
  <NOME>Paulo José Bastos</nome>
  ...
</ENTRADA>
...
```

Como poderá concluir mais tarde, nem esta é a maneira mais produtiva de associar imagens a documentos XML, nem este tipo de atributo é muito usado, estando a ser discutida a sua permanência na norma XML.

ENTITIES

Um atributo do tipo `ENTITIES` contém o nome de várias entidades declaradas algures no DTD.

Imagine que estávamos a tratar documentos que continham apresentações. Uma apresentação é um conjunto de *slides*. Cada *slide* é uma imagem declarada como uma entidade no DTD. Algures no documento, temos um elemento `apresentacao` com um atributo `slides` do tipo `ENTITIES` que indica a ordem pela qual os *slides* deverão ser visionados:

```
...
<apresentacao
  id="ap3"
  slides="slide2 slide5 slide6 slide7"/>
...
```

NOTATION

Este é o tipo de atributo menos utilizado e que, provavelmente, deixará de ser utilizado brevemente. O tipo `NOTATION` é usado para associar um tipo de dados externo (normalmente binário: gif, jpeg, mpeg, ...) a um determinado atributo ou elemento. Dada a sua pouca utilidade e utilização, não prolongaremos mais a sua discussão.

NMTOKEN

Os valores de atributos deste tipo são identificadores no contexto do XML: são constituídos por uma única palavra (não pode conter espaços); e podem ser uma mistura de letras, dígitos, hífens, *underscores* e pontos.

Este tipo de atributo é semelhante ao tipo `ID` (lexicamente) só que, em termos semânticos, não tem a restrição de unicidade.

De qualquer modo, é um dos tipos de atributo a cair rapidamente em desuso.

NMTOKENS

Quando o valor de um atributo precisa de ser composto por uma lista de `NMTOKEN` separados por espaço, usa-se este tipo.

À semelhança do anterior, é um tipo a cair em desuso. Normalmente, é utilizado em aplicações onde se coloca a informação nos atributos em lugar de a colocar nos elementos. Como iremos ver quando estivermos a discutir o processamento de documentos XML, esta é uma das situações a evitar.

Considere de novo o exemplo da agenda de contactos (ex. 23), e suponha que a cada entrada estará associada uma *alcunha*, para um tratamento mais pessoal. Em lugar de colocar essa informação num elemento, vamos colocá-la num atributo. Como uma *alcunha* pode ter mais do que uma palavra, vamos declarar este atributo com o tipo `NMTOKENS`:

```
<!ELEMENT ENTRADA (NOME, EMAIL, TELEFONE)>

<!ATTLIST ENTRADA
            IDENT ID #REQUIRED
            ALCUNHA NMTOKENS
            TIPO (pessoa|empresa|instituicao) "pessoa"
            FOTO ENTITY>
```

A única diferença entre este tipo e o `CDATA` são alguns sinais de pontuação que podem aparecer neste último.

Uma instância de uma `ENTRADA` com `alcunha` poderia ser:

```
... <ENTRADA IDENT="e7" ALCUNHA="trinca espinhas">
  <NOME>Paulo José Bastos</nome>
  ...
</ENTRADA> ...
```

2.3.3 Classes

A classe de um atributo interfere directamente com a tarefa de validação e de processamento, indicando qual o comportamento esperado do utilizador relativamente ao atributo: se o atributo é opcional, se é obrigatório ou se tem um valor fixo.

A norma XML define três classes de atributos:

#IMPLIED

O atributo é opcional. Cada instância do elemento ao qual o atributo está associado poderá indicar ou não, explicitamente, um valor para o atributo. Caso não tenha um valor associado, o valor assumido será indefinido ou inexistente. Por exemplo, a próxima declaração especifica dois atributos opcionais para o elemento `imagem`: a `altura` e a `largura`.

```
<!ATTLIST imagem
            altura CDATA #IMPLIED
            largura CDATA #IMPLIED>
```

#REQUIRED

O atributo é obrigatório. Nenhum valor é fornecido, mas o utilizador terá de providenciar um para cada instância.

Por exemplo, cada uma das entradas da agenda de contactos tem de ter um identificador e cada referência tem de ter o atributo `ref` instanciado.

```
<!ATTLIST ENTRADA IDENT ID #REQUIRED>
<!ATTLIST REFERENCIA REF IDREF #REQUIRED>
```

#FIXED

O atributo é constante e imutável e o seu valor será sempre o que estiver na declaração à frente da palavra-chave `#FIXED`. Se o atributo for instanciado, terá de ser com um valor igual ao declarado. Caso não seja instanciado, o sistema de processamento assumirá o valor declarado.

Por exemplo, suponha que numa dada aplicação se pretende associar um tipo de dados mais forte a um dado elemento, i. e., pretende-se indicar que os dados armazenados no elemento são mais do que uma simples *string*. Neste caso, o elemento `data` irá armazenar datas e pretende-se indicar que o conteúdo do elemento é mais do que uma *string*, é uma data que segue um determinado formato. Como o DTD não tem um meio explícito para o fazer, vamos fazê-lo recorrendo a um atributo com um valor por omissão `#FIXED`.

```
<!ELEMENT data ...>
<!ATTLIST data
            tipo CDATA #FIXED "data">
```

Desta maneira, qualquer processador vai assumir que todos elementos `data` que aparecerem instanciados num documento têm um atributo `tipo` com o valor `"data"`, podendo usar esse valor para parametrizar as acções que irão executar. Mais tarde, quando estivermos a discutir a transformação de documentos XML, iremos ver como tirar partido deste tipo de atributos.

A indicação da classe na declaração de um atributo é opcional. Se não for especificada as ferramentas de processamento tratarão o atributo como se este pertencesse à classe `#IMPLIED`.

2.3.4 Valores por omissão

Além de especificar um tipo e uma classe para o atributo, uma declaração `ATTLIST` inclui também a especificação de um valor por omissão. O valor por omissão é o valor que será usado para aquele atributo se o utilizador não especificar um na instância.

O valor por omissão é uma *string*. Por exemplo, e voltando à agenda de contactos, tínhamos a seguinte declaração:

```
<!ATTLIST ENTRADA
...
            TIPO (pessoa|empresa|instituicao) "pessoa">
```

O atributo TIPO é do tipo enumerado e da classe #IMPLIED, visto que nada é indicado sobre a classe, e tem um valor por omissão que é uma *string*, pessoa. Os outros tipos de atributo, como CDATA ou NMTOKEN, também podem ter um valor por omissão que é também uma constante.

Relativamente aos atributos não há muito mais a dizer. Muitas vezes, a dúvida entre a declaração de um componente como atributo ou como elemento subsistirá e só a prática poderá ajudar o utilizador na decisão.

2.4 Associação de um DTD a um documento

Depois de vermos como especificar um DTD, vamos agora discutir as várias possibilidades que há para associar um DTD a um documento.

Um documento XML completo tem de incluir uma referência para o DTD com o qual tem de ser comparado. Esta referência é especificada no início do documento através de uma declaração DOCTYPE:

```
<!DOCTYPE agenda SYSTEM "agenda.dtd">
```

Esta declaração especifica que o elemento raiz do documento é agenda e que o DTD pode ser encontrado no sistema no ficheiro *agenda.dtd*.

A indicação do nome do ficheiro onde se encontra o DTD pode assumir a forma de um URL completo:

```
<!DOCTYPE agenda
       SYSTEM "http://xml.di.uminho.pt/DTDs/agenda.dtd">
```

Esta declaração não pode aparecer livremente num documento XML. Ela deve aparecer sempre após a declaração XML e antes do elemento raiz. No caso da agenda teríamos:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE agenda SYSTEM "agenda.dtd">
<AGENDA>
  ...
</AGENDA>
```

A declaração DOCTYPE é uma reminiscência do passado, do SGML ([**?**, **?**]). Como tal, a sua sintaxe difere um pouco da sintaxe usada no XML. Esta declaração pode ainda assumir uma outra forma que era muito comum no SGML: pode usar um catálogo e um sistema de identificadores públicos para se referir ao DTD. No entanto, esse mecanismo foi praticamente colocado de lado no XML e a maioria dos processadores de XML trabalha com DTD referidos por um URL.

2.4.1 Redefinição parcial de um DTD

A sintaxe da declaração DOCTYPE pode ser um pouco mais complicada. A declaração pode ter um bloco extra onde é possível redefinir partes do DTD. A ideia subjacente é a de que um DTD define uma classe genérica de documentos e poderá haver documentos nessa classe com necessidades específicas muito particulares que tenham de ser especificadas à parte apenas para esse documento.

O bloco de redefinições surge no fim entre parêntesis rectos. Por exemplo, suponha que, no caso da nossa agenda de contactos, queremos acrescentar um campo extra opcional ao elemento ENTRADA, o URL:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE agenda SYSTEM "agenda.dtd" [
  <!ELEMENT ENTRADA (NOME, EMAIL, TELEFONE, URL?)>
  <!ELEMENT URL (#PCDATA)>
]>
<AGENDA>
  ...
</AGENDA>
```

Nesta instância de agenda, as entradas poderão ter um URL associado.

A redefinição de um DTD pode ser levada ao extremo e, nesse caso, podemos definir completamente o DTD no bloco de redefinições. Novamente, para a agenda teríamos:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE agenda [
<!ELEMENT AGENDA (ENTRADA | GRUPO)+>
<!ELEMENT ENTRADA (NOME, EMAIL, TELEFONE)>
...
]>
<AGENDA>
  ...
</AGENDA>
```

Note que, nesta situação, já não há indicação de onde procurar o DTD, pois este está completamente definido no bloco de redefinições.

Esta não é a situação normal. Um DTD define uma classe de documentos e, como tal, não deve ficar "agarrado" a uma instância documental. No entanto, esta é uma situação que poderá ser útil em determinadas circunstâncias: quando estamos a desenvolver o DTD ou quando queremos enviar um documento com o respectivo DTD a alguém (em vez de enviarmos dois ficheiros e confiarmos nos conhecimentos da pessoa para saber o que há-de fazer com eles, podemos enviar um só ficheiro e, desta maneira, as ferramentas que irão processar o documento tratarão da validação, sendo este processo transparente para o utilizador).

Neste capítulo, descreveu-se a criação e a utilização de um DTD. Um DTD serve para especificar a estrutura de um documento XML em termos das suas componentes lógicas.

Um documento pode, no entanto, também estar organizado em termos físicos e não ser o monobloco a que as pessoas estão habituadas. No próximo capítulo, vamos ver quais os mecanismos disponíveis em XML para organizar fisicamente um documento.

3 XML Path Language (XPath)

No capítulo anterior, viu-se que as transformações XSLT são determinadas pela estrutura lógica do documento XML (ADA) que se quer transformar e pelas `templates` na `stylesheet` XSLT. As expressões XPath incluídas nas `templates` são as responsáveis pela definição de qual a `template` que é aplicada a determinado nodo. Por isso, o XPath é um assunto central que é preciso dominar para se poder trabalhar com XSL.

Em termos funcionais, a utilização do Xpath em XSL pode ter uma série de objectivos:

- Selecção de nodos para processamento;
- Especificação de condições permitindo diferentes modos de processamento de um mesmo nodo;
- Geração de texto a ser incluído na árvore final (ADA resultante).

Muitas das operações envolvidas nestes objectivos envolvem escolhas baseadas no valor de um elemento, no valor de um atributo ou numa série de outros factores.
O XPath fornece-nos a funcionalidade para fazer aquelas escolhas numa sintaxe que não é XML.

O XPath é uma sintaxe usada para descrever partes de um documento XML. Com o XPath, é possível referenciar o primeiro elemento `aluno`, o atributo `regime` de `aluno`, todos os elementos `nome` cujo conteúdo contém `José` e muitas outras variações. Uma *stylesheet* XSLT usa expressões XPath em dois atributos, `match` e `select`, que estão associados a vários elementos do XSLT e que permitem guiar a transformação do documento.

O XPath foi desenvolvido para ser utilizado como valor de um atributo num documento XML. A sua sintaxe é uma mistura da linguagem de expressões, normalmente utilizada noutras linguagens de programação (como $23 - 7 * contador$) com a linguagem para a especificação do caminho numa estrutura de directorias como a usada nos sistemas *Unix* ou *Windows* (como $/poema/titulo$). Adicionalmente, o XPath fornece ainda um conjunto de funções para manipulação de texto, Namespaces, e outras funcionalidades normalmente necessárias na transformação de um documento.

Outro pormenor importante que é preciso reter é que o XPath trabalha com a versão do documento XML processada pelo *parser*, isto é, as entidades no texto foram expandidas e as secções especiais de texto (`CDATA`) foram convertidas para texto. Assim, não há maneira de saber se um nodo textual na ADA estava no documento original como uma entidade ou uma secção `CDATA`.

3.1 O Modelo de Dados do XPath

Do ponto de vista do XPath, um documento XML é uma ADA, ou seja, uma árvore de nodos. Para o XPath, há sete tipos de nodos:

- o nodo raiz (um por documento);
- nodos elemento;
- nodos atributo;
- nodos texto;
- nodos comentário;
- nodos instrução de processamento;

- nodos *namespace*.

Vamos reutilizar a seguinte instância do poema (já apresentado no ex. 24), para exemplificar os vários tipos de nodos:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- Poema anotado de acordo com poema.xsd -->
<poema tipo="soneto">
  <titulo>"Soneto Já Antigo"</titulo>
  <autor>(Álvaro de Campos)</autor>
  <corpo>
    <quadra>
      <verso>Olha, <nome>Daisy</nome>: quando eu morrer
             tu hás-de</verso>
      <verso>dizer aos meus amigos aí de
             <lugar>Londres</lugar>,</verso>
      <verso>embora não o sintas, que tu escondes</verso>
      <verso>a grande dor da minha morte. Irás de</verso>
    </quadra>
    <quadra>
      ...
    <terno>
      <verso>embora não o saibas, que morri...</verso>
      <verso>Mesmo ele, a quem eu tanto julguei
             amar,</verso>
      <verso>nada se importará... Depois vai dar </verso>
    </terno>
    <terno>
      ...
    </terno>
  </corpo>
  <data>(1922)</data>
</poema>
```

3.1.1 Nodo raiz

Em XPath, o nodo raiz é o nodo que contém o documento inteiro. No nosso caso, o nodo raiz contém o elemento `poema`, mas não é o elemento `soneto`. Numa

expressão XPath, o nodo raiz é sempre representado por '/'.

Ao contrário dos outros nodos, o nodo raiz não tem um nodo pai e tem pelo menos um nodo filho, o nodo que representa o documento. O nodo raiz também pode conter comentários e instruções de processamento que estejam fora do elemento que representa o documento. Na instância do poema, há uma instrução de processamento de nome `xml` e um comentário (posicionado antes do elemento `poema`) que estão nestas circunstâncias e que são filhos do nodo raiz. O valor textual do nodo raiz corresponde à concatenação de todos os nodos texto existentes na sua descendência.

3.1.2 Nodos elemento

Todos os elementos no documento XML original são representados por um nodo. No caso do poema, a árvore terá nodos deste tipo para o elemento `poema`, `titulo`, `autor`, `corpo`, etc. Os filhos de um nodo elemento podem ser nodos texto, nodos elemento, nodos comentário e nodos instrução de processamento que ocorrem nesse elemento no documento original. O valor textual de um nodo elemento é a concatenação do texto deste nodo e de todos os seus filhos pela ordem em que estes aparecem no documento. Como já foi referido, todas as referências a entidades foram expandidas; não se pode manipular entidades gerais ou carácter em XPath.

3.1.3 Nodos atributo

Um nodo atributo tem sempre um pai que é um nodo elemento. No nosso exemplo, o nodo elemento correspondente ao elemento `poema` é pai de um nodo atributo com nome `tipo` e valor `soneto`.

Este tipo de nodos tem, no entanto, algumas funcionalidades distintas dos outros tipos de nodos:

- Apesar de um nodo elemento ser o pai dos seus nodos atributos, estes não são filhos do seu pai. Os filhos de um nodo elemento são os elementos texto, elemento, comentário e instrução de processamento contidos no elemento correspondente do documento original. Se se quiser seleccionar os nodos atributo tem de se indicar explicitamente. Mais tarde veremos que tratar os

elementos e os atributos separadamente dá jeito e é o que normalmente se pretende fazer.

- Em circunstâncias normais, o processador de XPath cria um nodo atributo para todos os atributos instanciados no documento original e para os atributos que tenham um valor por omissão declarado no DTD ou Schema (isto, claro, só se o processador tiver capacidade para analisar um DTD ou Schema).

3.1.4 Nodos texto

Os nodos texto são os mais simples, apenas contêm o texto do elemento correspondente. Se no documento original existirem referências a entidades estas serão resolvidas antes de o nodo ser criado, um nodo texto contém apenas texto puro. Além disso, um nodo texto contém o máximo de texto possível o que faz com que um nodo destes não tenha irmãos do seu tipo na árvore documental.

3.1.5 Nodos comentário

Um nodo comentário é também muito simples, contém apenas texto. O texto de um nodo comentário contém todo o conteúdo do comentário original excepto as marcas <! - e ->.

3.1.6 Nodos instrução de processamento

Um nodo instrução de processamento tem duas partes, um nome e um valor textual. O valor textual é tudo o que aparece a seguir ao nome excepto a marca de fecho ?>.

3.1.7 Nodos *namespace*

Os nodos deste tipo raramente são utilizados nas folhas de estilo XSL. Estes nodos existem principalmente para benefício do processador que tem de diferenciar elementos pertencentes a diferentes namespaces. As declarações de namespaces nas instâncias XML apesar de corresponderem tecnicamente a atributos, são tratadas em XPath como nodos do tipo namespace (uma declaração como:

`xmlns:autor="http://www.autor.pt"`, seria guardada num nodo do tipo `namespace`).

3.2 XPath como selector de nodos

Alguns comandos do XSLT utilizam um selector para seleccionar um nodo ou um conjunto de nodos que serão alvo de uma transformação. O selector é uma expressão XPath e esta é a utilização mais comum do XPath em XSLT.

3.2.1 Contexto

Outro conceito importante em XPath é o contexto. Tudo aquilo que se faz em XPath é relativo ao contexto. Se estabelecermos uma analogia com um sistema de ficheiros e directorias num sistema operativo como o *Unix* ou o *Windows*, o contexto corresponderá à directoria corrente. Nesta analogia, podemos ver a ADA como uma árvore de directorias e o conteúdo dos seus nodos como ficheiros.

Neste livro, o contexto é um nodo da árvore documental abstracta a partir do qual uma expressão XPath é avaliada.

Quando se trabalha num sistema operativo, a maior parte das vezes, executam-se comandos na directoria corrente. O mesmo acontece em XSLT, muitas vezes, o selector corresponderá ao contexto actual, o que permitirá a sua supressão.

3.2.2 Selectores Simples

A seguir, descrevem-se alguns dos operadores mais simples que podem ser usados em expressões XPath:

'/' — no início, selecciona o nodo raiz da ADA; no meio de uma expressão, funciona apenas como separador de dois níveis da árvore.

'.' — representa o contexto actual; selecciona o nodo corrente.

'..' — selecciona o pai do nodo correspondente ao contexto actual.

Estes operadores podem combinar-se em expressões XPath mais complexas.
Por exemplo:

`/poema/corpo/quadra` — esta expressão selecciona todos os nodos do *tipo* `quadra` que são filhos de nodos `corpo`, que são filhos do nodo `poema` (só há um, é o elemento raiz do documento), que, por sua vez, é filho do nodo raiz da ADA (`''`).

`./verso/nome` — esta expressão selecciona todos os nodos do *tipo* `nome` que são filhos de nodos `verso`, que, por sua vez, são filhos do nodo corrente ou contexto actual.

`../titulo` — esta expressão selecciona todos os nodos do *tipo* `titulo` que são filhos do nodo pai do nodo corrente ou contexto actual.

Eis algumas expressões XPath, tendo como base de aplicação o poema, e uma breve descrição de cada uma:

`/` — Selecciona o nodo raiz da ADA que, no caso do poema, tem três nodos filho: um nodo elemento (`poema`), um nodo instrução de processamento (contendo a declaração XML) e um nodo comentário.

`/poema/corpo/quadra/verso` — Selecciona todos os versos das quadras.

3.2.3 Selectores Relativos e Selectores Absolutos

Há dois tipos de expressões em XPath, absolutas e relativas. Uma expressão absoluta é sempre iniciada pela indicação do nodo raiz da ADA: `/`. Uma expressão absoluta pode ser avaliada em qualquer momento mesmo quando o contexto actual não é a raiz do documento. Uma expressão relativa é aquela que tem em conta o contexto actual.

Há vantagens e desvantagens associadas a estes dois tipos de expressões. Enquanto uma expressão absoluta facilita a vida ao processador de XSL, também complica a vida ao programador dificultando a reutilização. Para uma expressão relativa, podemos dizer exactamente o inverso. Esta questão ilustra-se com o exemplo seguinte.

Exemplo 26: Expressões absolutas e expressões relativas

Considere de novo a instância do poema apresentada no início do capítulo e observe as seguintes expressões XPath:

```
(e1)   /poema/corpo/quadra/verso

(e2)   verso
```

A primeira expressão é absoluta e selecciona apenas os elementos `verso` que são filhos de elementos `quadra` que, por sua vez, são filhos de elementos `corpo` que terão de ser filhos do elemento principal `poema`.

A segunda expressão é relativa e vai seleccionar os elementos `verso` sempre que estes possam ser seleccionados, o que equivale a dizer que estes elementos serão seleccionados sempre que o contexto actual corresponder a um elemento `quadra` ou `terno`.

Como veremos adiante a segunda expressão é mais universal e se os versos das quadras e dos ternos tiverem de sofrer a mesma transformação, esta expressão permitirá seleccioná-los de uma só vez.

3.2.4 Seleccionar: para além de elementos

Até agora, o resultado das expressões utilizadas foi sempre um elemento ou um conjunto de elementos. Mas, como vimos atrás, um documento XML contém outras coisas para além dos elementos. Nas secções seguintes, mostra-se como é que se podem seleccionar e manipular essas outras entidades.

Selecção de Atributos

Para seleccionar um atributo, usa-se o carácter `@` como prefixo do nome do atributo. No nosso caso do poema, selecciona-se o tipo deste com a seguinte expressão: `/poema/@tipo`.

Se o contexto actual correspondesse ao nodo poema, a expressão de selecção podia ser reduzida à seguinte expressão relativa: `@tipo`

Selecção do texto dum elemento

Para seleccionar o conteúdo textual de um determinado elemento, é preciso utilizar uma função selectora do XPath de nome `text()` que devolve o conteúdo textual de um determinado nodo do tipo elemento (lembrar que, se o nodo ao qual a função é aplicada for um nodo intermédio, o texto devolvido corresponde à concatenação do conteúdo de todos os nodos do tipo texto pertencentes à sua descendência).

Eis algumas expressões possíveis para o caso do poema:

/poema/titulo/text() selecciona o conteúdo textual do título do poema.

/poema/text() selecciona o conteúdo textual do poema inteiro.

Selecção de Comentários e Instruções de Processamento

O XPath possui mais algumas funções de selecção, duas das quais permitem seleccionar nodos do tipo comentário e nodos do tipo instrução de processamento: `comment()` e `processing-instruction()`. A utilização destes selectores será sempre muito esporádica pois, normalmente, a informação colocada em comentários ou instruções de processamento tem um fim muito específico. Mas uma utilização possível seria a intenção de visualizar os comentários no documento transformado.

Voltando ao nosso caso de estudo:

`/comment()` selecciona o comentário inicial, que, como está fora do elemento principal `poema`, é filho da raiz.

`/processing-instruction()` selecciona a declaração XML.

3.2.5 Selectores Complexos

Além dos selectores já discutidos, o XPath possui quatro selectores mais complexos:

'*' — O selector representado pelo asterisco selecciona todos os elementos no contexto corrente. Este selector selecciona apenas nodos do tipo elemento. Nodos do tipo atributo, texto, comentário e instrução de processamento não são seleccionados.

'@*' — A expressão composta pelo selector de atributo e pelo asterisco selecciona todos os atributos no contexto actual.

'node()' — A função selectora `node()` selecciona todos os nodos no contexto actual independentemente do tipo.

'//' — A dupla barra de divisão, quando utilizada no meio de uma expressão XPath, indica que entre as duas barras podem ocorrer zero ou mais elementos. Por exemplo, a expressão "`//verso`" selecciona todos os versos do poema quer sejam filhos de `quadra` ou `terno`.

A seguir apresentam-se alguns exemplos que usam combinações dos vários selectores apresentados:

/poema/corpo/*/verso selecciona os versos independentemente de serem filhos de `quadra` ou `terno`.

//nome selecciona todos os nodos elemento de nome "nome" em qualquer ponto da árvore documental.

//@* selecciona todos os atributos de todos os nodos.

//* selecciona todos os nodos do tipo elemento.

//node() selecciona todos os nodos de qualquer tipo.

//comment() selecciona todos os comentários presentes no documento.

//quadra/text() selecciona o conteúdo textual das duas quadras.

Com estes exemplos, encerra-se a primeira parte do XPath. A segunda parte, inicia-se já na secção seguinte e introduz um conceito novo e importante: o eixo de navegação na árvore documental abstracta.

3.3 Eixos de Navegação

Até agora, foi possível seleccionar elementos, atributos, texto, comentários, com algumas expressões XPath simples, na maioria dos casos, explorando a relação pai–filho entre elementos.

No entanto, em muitas aplicações, é necessário seleccionar nodos segundo uma perspectiva diferente, como por exemplo:

- Todos os nodos ancestrais do nodo corrente (o pai, o avô, ...).
- Todos os nodos descendentes do nodo corrente.
- Todos os nodos irmãos precedentes (à esquerda) ou sequentes (à direita) do nodo corrente.

Para este tipo de seleccções, o XPath disponibiliza um mecanismo que designaremos por `eixo de navegação`. Ao todo, o XPath tem treze eixos de navegação que se descrevem e ilustram com exemplos a seguir.

Este conceito é fulcral no XPath e é conveniente que seja bem explicado. Para isso, podemos afirmar que qualquer expressão XPath usa uma determinada relação entre elementos para "encaminhar" a selecção ao longo duma árvore documental. É a esta relação subjacente à expressão que chamamos `eixo de navegação`. Nas expressões discutidas até ao momento, este conceito já estava presente, só que, uma vez que a relação utilizada era a de pai–filho, a mais comum, o XPath, permite-nos abreviar omitindo o `eixo de navegação` para este caso.

Para se utilizar um `eixo de navegação` numa expressão XPath, indica-se o nome do eixo seguido de dois sinais de dois pontos (’::’) seguido do nome do elemento que se quer seleccionar. Por exemplo e atendendo ao ponto anterior sobre as expressões abreviadas, a expressão `quadra/verso` é equivalente a `child::quadra/child::verso`.

A seguir, listam-se os eixos de navegação disponíveis no XPath, descrevendo-se sucintamente cada um deles:

child Selecciona os filhos do nodo corrente. Como já foi dito, a omissão do eixo de navegação corresponde à indicação implícita deste eixo. Nos nodos filho do nodo corrente estão incluídos os nodos elemento, comentário, instrução de processamento e textuais. Os nodos do tipo atributo e `namespace` não são abrangidos pela selecção deste eixo.

parent Selecciona o nodo pai do nodo corrente, se existir (se estivermos posicionados na raiz o resultado devolvido é uma lista vazia de nodos). Este eixo pode ser abreviado com a utilização do selector dois pontos ('..'). A expressão `parent::verso` e a expressão `../verso` são equivalentes.

self Selecciona o nodo corrente. Este eixo pode também ser abreviado com a utilização do ponto ('.'). A expressão `self::*` e a expressão `.` são equivalentes.

attribute Selecciona os atributos do nodo corrente. Se o nodo corrente não for do tipo elemento, o resultado será uma lista vazia. Este eixo também pode ser abreviado com o selector de atributos já apresentado `@`. A expressão `/poema/attribute::tipo` e a expressão `/poema/@tipo` produzem o mesmo resultado.

ancestor Selecciona todos os ancestrais do nodo (pai, avô, ...) até à raiz.

ancestor-or-self Este eixo tem um comportamento igual ao anterior, só que à lista resultante acrescenta o nodo corrente.

descendant Selecciona todos os descendentes do nodo corrente (os filhos, os netos, ...). Como já vimos noutras situações, são abrangidos por esta selecção os nodos do tipo elemento, comentário, instrução de processamento e texto. Os nodos do tipo atributo e `namespace` não são abrangidos.

descendant-or-self Este eixo tem um comportamento igual ao anterior, só que à lista resultante acrescenta o nodo corrente.

preceding-sibling Selecciona todos os irmãos precedentes do nodo corrente. Por outras palavras, selecciona todos os nodos que têm o pai do nodo corrente e que aparecem antes do nodo corrente no documento XML. Se o nodo corrente for do tipo atributo ou `namespace`, o resultado será uma lista vazia.

following-sibling Selecciona todos os irmãos sequentes do nodo corrente. Por outras palavras, selecciona todos os nodos que têm o pai do nodo corrente e que aparecem depois do nodo corrente no documento XML. Se o nodo corrente for do tipo atributo ou `namespace`, o resultado será uma lista vazia.

preceding Selecciona todos os nodos que aparecem antes do nodo corrente no documento, excepto os nodos ancestrais e os nodos atributo e `namespace`.

following Selecciona todos os nodos que aparecem depois do nodo corrente no documento, excepto os nodos descendentes e os nodos atributo e `namespace`.

namespace Selecciona os nodos `namespace` do nodo corrente. Caso o nodo corrente não corresponda a um elemento, o resultado será uma lista vazia.

Nos restantes capítulos do livro, surgirão exemplos práticos onde estes eixos de navegação serão utilizados para resolver problemas práticos. No entanto, ficam aqui alguns exemplos didácticos de utilização de expressões com eixos sobre o poema. Para cada expressão e_i é indicado o resultado r_i.

```
(e1.1) /child::poema/child::corpo/child::quadra/child::*
(e1.2) /poema/corpo/quadra/*

  (r1) são seleccionados, para cada quadra,
       todos os nodos filho


(e2.1) /poema/corpo/quadra/child::lugar
(e2.2) /poema/corpo/quadra/lugar

  (r2) são seleccionados, para cada quadra,
       todos os nodos filho de nome lugar

(e3.1) parent::lugar
(e3.2) ../lugar

  (r3) sempre que o nodo corrente fôr lugar,
       selecciona o nodo pai
```

```
(e4)   quadra/ancestor::*

  (r4) sempre que o nodo corrente fôr quadra,
       selecciona os nodos ancestrais, o resultado
       é a lista: [corpo,poema,/]


(e5)   quadra/preceding-sibling::*

  (r5) se o nodo corrente fôr a segunda quadra,
       o resultado será a primeira quadra


(e6)   quadra/following-sibling::*

  (r6) se o nodo corrente fôr a segunda quadra,
       o resultado será composto pelos dois ternos
```

Na próxima secção, vamos tratar outro dos conceitos fundamentais do XPath, os predicados. Estes vão permitir que o utilizador especifique critérios de selecção muito específicos.

3.4 Predicados

Um predicado pode ser definido como um filtro que restringe os nodos seleccionados por uma expressão XPath. Os predicados são avaliados em tempo de execução e dão como resultado um valor booleano (verdadeiro ou falso). Se, para um determinado nodo, o resultado da avaliação do predicado for verdadeiro, esse nodo é seleccionado.

Em termos sintácticos, um predicado é especificado dentro de parêntesis rectos. Por exemplo:

```
//quadra/verso[2]
```

Esta expressão selecciona o segundo verso das quadras do poema.

Há várias coisas, que se discutem a seguir, que podem surgir num predicado.

Números – Um predicado constituído apenas por um número selecciona os nodos que têm essa posição particular. Por exemplo, a expressão

```
//quadra[1]/verso[2]
```

selecciona o segundo verso da primeira quadra do poema. Na realidade, este tipo de predicados corresponde a uma abreviatura do teste da posição do elemento. Sem abreviaturas a expressão acima seria:

```
//quadra[position()=1]/verso[position()=2]
```

Atributos – Um predicado constituído por uma selecção de atributo é verdadeiro se esse atributo existir no elemento corrente. Eis alguns exemplos comuns:

```
/poema[@tipo]

  selecciona o poema mas só se este tiver o
  atributo tipo instanciado


//*[@*]

  selecciona qualquer elemento que tenha um
  qualquer atributo instanciado
```

Funções – Um predicado pode conter invocações de funções do XSLT (serão discutidas na próxima secção). Por exemplo:

```
quadra[last()]

  selecciona a última quadra do contexto corrente


verso[position() mod 2 = 0]

  selecciona os versos do contexto corrente que se
  encontram nas posições pares
```

Combinadores de predicados – O XPath tem ainda alguns operadores que permitem combinar predicados. É o caso dos operadores booleanos `and` e `or` e do operador de união `|`. Eis alguns exemplos:

```
verso[(position() mod 2 = 0) or (position()=last())]
verso[1|3]
```

Na próxima secção, depois de se apresentar algumas funções do XPath, apresentam-se alguns predicados mais complexos.

3.5 Funções

Até ao momento, é possível concluir que o XPath é uma linguagem poderosa no que concerne à selecção de nodos numa árvore documental abstracta. No entanto, existem expressões de selecção que não é possível especificar com os mecanismos discutidos até agora. Para essas situações a norma propõe a utilização de algumas funções (quase todas implementadas nos processadores de XSL discutidos neste livro).

As funções do XPath podem ser divididas em quatro categorias. A seguir apresentam-se, para cada categoria, a lista de funções existentes juntamente com uma breve descrição.

Funções para manipulação de listas de nodos – Agrupamos nesta categoria as funções que realização cálculos sobre a árvore documental abstracta.

position() – Dá como resultado um número correspondente à posição do nodo na árvore documental ou, no caso de estar a ser aplicada ao resultado duma operação de ordenação, a posição na lista ordenada.

last() – Dá como resultado um número correspondente ao total de nodos existentes no nível da árvore do nodo corrente (se for usada num predicado permite seleccionar o último nodo desse nível uma vez que o número de nodos de um determinado nível é igual à posição do último nodo nesse nível).

count(xpath-exp) – Dá como resultado o número de nodos seleccionados pela expressão XPath dada como argumento. Por exemplo:

```
count(//verso)
  conta o número de versos no poema
```

```
count(//*)
  conta o número de elementos no poema

count(quadra[1]/ancestor::*)
  conta o número de ancestrais da primeira quadra
  dá o nível da árvore em que a quadra se encontra
```

id(identificador) – Dá como resultado o nodo que tem um atributo do tipo `ID` com valor igual a `identificador`.

Funções para manipulação de *strings* – Nesta categoria, estão as funções que permitem manipular texto. Como o utilizador experimentado poderá constatar muitas delas são habituais nas linguagens de programação ou em sistemas operativos da família do Unix.

concat(str1,str2,...) – Dá como resultado uma *string* resultante da concatenação das várias strings argumento.

starts-with(str1,str2) – Retorna um valor booleano verdadeiro se `str1` começar por `str2`.

contains(str1,str2) – Retorna um valor booleano verdadeiro se `str1` contiver `str2`.

substring(str,num,comp) – Retorna uma *string* que se extrai da *string* argumento começando na posição `num` e de comprimento `comp`.

substring-before(str1,str2) – Retorna uma *substring* da primeira *string* argumento composta pelos caracteres anteriores à primeira ocorrência da segunda *string* argumento.

substring-after(str1,str2) Retorna uma substring da primeira string argumento composta pelos caracteres posteriores à primeira ocorrência da ssegunda string argumento.

string-length(str) – Retorna o número de caracteres na *string*.

normalize-space(str) – Retorna a *string* argumento com o espaço normalizado: são retirados os espaços do início e do fim, e, todas as sequências de caracteres brancos no meio da *string* são substituídas por um único espaço em branco.

translate(str1,str2,str3) – Retorna a primeira *string* argumento (`str1`) com as ocorrências de caracteres de `str2` pelo respectivo carácter (na mesma posição) de `str3`.

Funções booleanas – Nesta categoria, agruparam-se as funções que produzem um resultado booleano como resultado.

boolean(arg) – Converte o argumento, que pode ser qualquer coisa, num valor booleano.

not(bool-exp) – Retorna o valor booleano inverso da expressão booleana argumento.

true() – Retorna o valor booleano verdadeiro.

false() – Retorna o valor booleano falso.

lang(str) Retorna verdadeiro se a língua do documento for igual à língua indicada no argumento.

string(arg) Converte o seu argumento, que pode ser de qualquer tipo, numa *string*. Esta função tem uma aplicação curiosa e útil: quando aplicada a um elemento cujo conteúdo esteja vazio, devolve o valor booleano falso o que nos fornece uma maneira de testar elementos vazios.

Funções numéricas – Por fim, nesta categoria, agruparam-se as funções que nos permitem realizar cálculos aritméticos com o conteúdo dos elementos.

number(arg) – Converte o seu argumento, que pode ser de qualquer tipo, num número. Se o argumento for omitido a função é aplicada ao nodo corrente.

sum(xpath-exp) – Retorna o resultado da soma da conversão para número de todos os elementos seleccionados pela expressão argumento. Se um dos nodos tiver um conteúdo que não seja número o valor retornado é `NaN` (*Not a Number*).

floor(num) – Retorna o maior inteiro menor ou igual ao argumento.

ceiling(num) – Retorna o inteiro mais pequeno que não é menor que o argumento.

round(num) – Retorna o inteiro mais próximo do argumento.

3.6 Exemplos de expressões XPath

A seguir, apresenta-se uma lista de exemplos, onde se combinam todos os conceitos, operadores e funções discutidos até aqui. Nos exemplos que se seguem, não

consideramos nenhuma instância XML em particular, por isso usamos nomes de elementos fictícios como AAA, BBB ou CCC.

/AAA – Selecciona o nodo filho da raiz com nome AAA.

/AAA/CCC – Selecciona nodos de nome CCC filhos do nodo principal AAA que é filho da raiz.

//BBB – Selecciona todos os nodos de nome BBB existentes na árvore documental.

//DDD/BBB – Selecciona todos os nodos de nome BBB que sejam filhos de nodos DDD posicionados em qualquer ponto da árvore documental.

/AAA/CCC/DDD/* – Selecciona todos os nodos filhos de nodos DDD que, por sua vez, são filhos de nodos CCC que são filhos do nodo principal AAA que é filho da raiz.

/*/*/*/BBB – Selecciona todos os nodos BBB posicionados no quarto nível da árvore documental.

//* – Selecciona todos os nodos do tipo elemento existentes na árvore documental.

/AAA/BBB[1] – Selecciona o primeiro filho com nome BBB do nodo AAA que é filho da raiz.

/AAA/BBB[last()] – Selecciona o último filho de nome BBB do nodo AAA que é filho da raiz.

//BBB[@ident] – Selecciona todos os nodos com nome BBB que tenham um atributo de nome `ident` instanciado.

//BBB[@*] – Selecciona todos os nodos com nome BBB que tenham pelo menos um atributo instanciado.

//BBB[not(@*)] – Selecciona todos os nodos com nome BBB que não têm nenhum atributo instanciado.

//BBB[@ident='b1'] – Selecciona todos os nodos com nome BBB que tenham um atributo de nome `ident` instanciado com o valor `b1`.

//BBB[normalize-space(@nome)='bbb'] – Selecciona todos os nodos com nome `BBB` que tenham um atributo de nome `nome` cujo valor normalizado (sem espaços no fim e no início e com as sequências de caracteres brancos reduzidas a um espaço em branco) seja igual a `bbb`.

//*[count(BBB)=2] – Selecciona todos os nodos da árvore documental que tenham exactamente dois filhos com nome `BBB`.

//*[count(*)=2] – Selecciona todos os nodos da árvore documental que tenham exactamente dois filhos.

//*[name()= BBB] – Selecciona todos os nodos da árvore documental com nome igual a `BBB`.

//*[starts-with(name(), 'B')] – Selecciona todos os nodos da árvore documental cujo nome se inicie por `B`.

//*[contains(name(),'C')] – Selecciona todos os nodos da árvore documental cujo nome contenha o carácter `C`.

//*[string-length(name())=3] – Selecciona todos os nodos da árvore documental cujo nome seja constituído por 3 caracteres.

//*[string-length(name()) < 3] – Selecciona todos os nodos da árvore documental cujo nome seja constituído por menos de 3 caracteres.

//BBB | **//CCC** – Selecciona todos os nodos na árvore documental com nome `BBB` ou `CCC`.

/descendant::* – Selecciona todos os nodos descendentes do nodo raiz.

//CCC/descendant::* – Selecciona todos os nodos descendentes de nodos com nome `CCC`.

//CCC/descendant::*/DDD – Selecciona todos os nodos descendentes de nodos com nome `CCC` e cujo nome seja `DDD`.

Termina aqui este capítulo sobre XPath. No próximo capítulo, introduz-se a construção de folhas de estilo XSLT e veremos como e onde é que o XPath se integra no processamento de um documento XML.