Anotação Estrutural de Documentos e sua Semântica

Especificação da Sintaxe, Semântica e Estilo para Documentos

José Carlos Leite Ramalho
Departamento de Informática - Escola de Engenharia - Universidade do Minho

Supervisão: Pedro Rangel Henriques

Anotação Estrutural de Documentos e sua Semântica: Especificação da Sintaxe, Semântica e Estilo para Documentos

por José Carlos Leite Ramalho e Supervisão: Pedro Rangel Henriques

Este documento descreve o trabalho realizado no âmbito da tese de doutoramento do autor.

O trabalho teve duas grandes linhas orientadoras. A estruturação de documentos, como a maneira de os tornar mais "ricos"e mais "vivos". E, a semântica dos documentos, desde a aparência visual até à interpretação (significado) do seu conteúdo. No fim, estas duas linhas acabaram por convergir na elaboração dum novo modelo de processamento documental.

Ao longo da dissertação, irá ser apresentada uma comparação de modelos de processamento documental, ou publicação electrónica; referir-se-á o processamento dos documentos normais, que são apenas textos, e dos documentos anotados, que têm uma estrutura lógica e um conteúdo. Esta análise será ilustrada com alguns casos práticos que se desenvolveram ao longo deste trabalho.

As vantagens dos documentos estruturados serão apresentadas e os passos para a implementação de um sistema de produção de documentos estruturados serão descritos.

A seguir, apresentar-se-á o conjunto de necessidades e requisitos actuais que se podem colocar a um sistema destes e analisar-se-á aquilo que se designou por "semântica dos documentos". As necessidades identificadas estão relacionadas com o problema da qualidade de conteúdos na publicação electrónica. A qualidade em publicações electrónicas pode ser analisada segundo vários parâmetros, desde o aspecto visual, o linguístico e literário, à correcção da informação (significado, semântica). A tecnologia existente permite de alguma forma automatizar e normalizar todos estes aspectos, excepto o último. Foi no desenvolvimento de uma solução para este problema que se centrou esta dissertação: como adicionar semântica estática (condições contextuais ou invariantes) aos documentos; e como processar esta semântica estática dum modo integrado com a tecnologia existente.

São apresentadas duas vias para a solução da especificação e processamento da semântica estática, a primeira segue uma aproximação via modelos abstractos, a outra, uma aproximação via gramáticas de atributos.

No fim, uma das soluções será escolhida e integrada num sistema (S4) que sugere um novo modelo de processamento para documentos estruturados e que explora alguns paradigmas novos neste contexto (adoptam-se para os documentos metodologias utilizadas nas linguagens de programação como consequência duma hipótese levantada pelo autor, da existência dum

paralelismo entre o processamento de documentos e o processamento das linguagens de programação), que vão desde a análise da informação até ao seu tratamento.

A dissertação inclui a apresentação dos passos seguidos na produção do seu próprio texto, uma vez que se adoptaram as soluções defendidas e nela apresentadas.

Dedicatória

À minha família, a Carmen, o David, e o pequeno Leonardo, que toleraram as minhas ausências e a minha obcessão durante um largo período de tempo.

Índice

Lista de Tabelas

Lista de Figuras

Lista de Exemplos

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao meu supervisor, o Professor Pedro Henriques, pela supervisão omnipresente e, mais importante ainda, a constante disponibilidade em discutir, analisar e reler resultados, permitindo um desfecho mais rápido e rico em termos de ideias.

Ao meu colega, José João, pelas discussões e parcerias donde resultaram ideias inovadoras, algumas delas concretizadas em ferramentas e publicações.

Aos meus co-autores, durante os últimos quatro anos, que colaboraram comigo na publicação de vários artigos e outras monografias: Pedro Henriques, José João, Jorge Rocha, Alda Lopes.

À Alda Lopes e ao Pedro Sousa o terem aceite fazer as suas teses de mestrado comigo. O seu trabalho deu um empurrão precioso na parte prática desta tese.

Aos alunos finalistas que comigo colaboraram na realização de vários projectos reais, dando-me o contacto com a realidade que tão necessário foi para a elaboração de algumas ideias: Nuno Figueiredo, Paulo Carvalho, Rogério Paiva, Salomé Ribeiro, Sónia Gaspar, Carla Lopes, Clara Oliveira, Armando Lemos, José Henrique Cerqueira, António Pedro Lopes.

Ao Norman Walsh, a prontidão com que sempre me acudiu na resolução de problemas com as stylesheets do Docbook.

Ao Sebastian Rahtz, a paciência que teve para comigo instalarmos o ambiente definitivo com que viria a ser produzida esta dissertação.

Capítulo 1. Introdução

Se observarmos de perto uma empresa, um estabelecimento de ensino, ou outra instituição pública ou privada, depressa concluímos que o produto de cada dia de trabalho resulta, directa ou indirectamente, na criação de documentos; cada membro de uma organização, na sua actividade diária, cria de alguma forma documentação. Pode ser uma carta, um memo, a planta de um edifício ou o manual de um produto. Quaisquer que sejam os documentos produzidos eles formam parte da história da instituição e são um importante componente da sua memória corporativa.

Algumas formas de informação são altamente estruturadas. De facto, algumas são tão estruturadas que permitem a sua representação numa forma tabular ou numérica. Informação sobre inventários, preços, empregados, é um exemplo deste tipo de informação. Hoje em dia, a maioria dos sistemas de gestão corporativa são desenhados para gerir informação relacional e muito estruturada (folhas de cálculo, BDs). Mas, surpreendentemente, estima-se que esta informação representa apenas 10% do total de informação disponível numa empresa. Assim surgem as seguintes questões: Que tipo de informação representa os outros 90%? Como se poderá tirar partido dela?

Estes 90% da informação correspondem a textos que são produzidos e circulam dentro da instituição. As metodologias relacionais são difíceis ou mesmo impossíveis de aplicar a texto. Pode-se então colocar a questão: Haverá alguma maneira de aceder ao potencial da informação mantida num formato textual? Se ele se mantiver numa forma simples, puramente sequencial, a solução parece difícil; a resposta recai, mais uma vez, na *estruturação* desses textos [Ken96].

Documentos estruturados são documentos que têm a sua estrutura explícita, as suas componentes estão identificadas. Se esta estrutura fôr definida formalmente, identificando as componentes e a maneira de a partir delas se construir o documento, torna-se possível estipular um conjunto de regras para a criação desses documentos. Por exemplo, as regras para um determinado manual podem estipular que o documento terá uma página de rosto, um índice, e um prefácio; esta parte inicial deverá ser continuada por no mínimo quatro capítulos; cada capítulo deverá ter um título seguido de texto ou ter ainda esse texto dividido em secções. Essas secções poderão ter também a sua estrutura, e assim por diante.

A maioria dos textos produzidos numa empresa são estruturados ou não estruturados? A probabilidade de eles terem sido produzidos de acordo com um

conjunto regras da empresa é muito grande. Isto faz com que eles tenham uma estrutura que lhes é inerente, o que não quer dizer que essa organização seja explícita.

Este problema, de tentar manter viva a informação correspondente aos 90% do património duma instituição dá-nos a primeira motivação para este trabalho.

O segundo mote, vem duma área relacionada: a publicação electrónica.

Nas últimas décadas, a Publicação Electrónica sofreu uma enorme evolução. O avançar da informática e das tecnologias a ela associadas tem levado a uma substituição gradual e efectiva do papel pelo suporte digital, magnético ou óptico (disquete, disco de computador, ou CDROM).

Nos últimos anos, a explosão da Internet (cada vez mais fácil e amplamente acessível), veio acelerar e aumentar ainda mais a produção documental em suporte digital, consolidando novos tipos e arquitecturas de informação: o hipertexto e a hipermedia [DD94].

Esta evolução trouxe com ela vários problemas e veio agravar outros já existentes. O mais grave foi, e é, o da proliferação de formatos proprietários e a incompatibilidade de cada um deles face aos outros. Com a Internet e a banalização da produção de CDROMs, para além dos vários sistemas tradicionais de arquivo e processamento de texto, os utilizadores passam a precisar de manter os seus documentos em mais formatos e suportes. Um documento leva tempo a produzir, consome espaço de arquivo pelo que, se a sua reutilização não fôr possível, o seu custo aumentará ainda mais. Por estas razões, a produção documental deve ser inteligente, de modo a que os documentos produzidos se mantenham vivos (ao fim de vários anos, em diferentes sistemas e depois de várias versões do software que os produziu) e que a sua reutilização, para os mais variados fins, seja possível.

Outro problema prende-se com a globalização da informação. Hoje a Internet é praticamente acessível a todos, tendo-se tornado um veículo apetecível para quem disponibiliza e para quem consome informação. Como resultado temos uma maior proliferação documental. Torna-se praticamente inviável colocar documentos puramente textuais na Internet. Os motores de pesquisa e de indexação teriam um trabalho infinito para procurar e encontrar fosse o que fosse. Neste contexto surgiram projectos como o "Dublin Core"[Hee96] ou o "TEI"[SB94] que visam resolver estes problema acrescentando meta-informação aos documentos. Meta-informação é informação sobre informação [Cap95], neste caso sobre documentos. Mais especificamente, trata-se duma espécie de registo bibliográfico

que se agrega a um documento de modo a disponibilizar facilmente informação como a data da sua criação, quem são os seus autores e até mesmo, uma memória descritiva do seu conteúdo.

Quando se fala de documentos e meta-informação, o "Dublin Core"é quase uma referência obrigatória. Isto resulta do facto de ter sido definido numa workshop que reuniu os especialistas nos mais variados ramos da informática e da meta-informação [WGMD]. No entanto, esta proposta de standard visa resolver apenas uma pequena parte do problema, o da localização de documentos na Internet. Não se preocupa com o seu conteúdo ou a sua estruturação, nem com a uniformização de formatos uma vez que aceita quase tudo desde Postscript [POSTSCRIPT] a SGML [Gol90].

Recentemente, uma das áreas que tem registado significativos contributos e que muito tem evoluído é a da representação abstracta de documentos, ou de objectos com a forma de documento. Aqui os problemas começam logo pela definição de documento. Para algumas pessoas um documento é apenas um registo textual enquanto para outras pode ser muita coisa desde texto até um ser vivo. O esforço para normalizar tem sido enorme e standards como o SGML [Her94], o Hytime [DD94], o XML [XML], e propostas como o DOM [DOM] e o RDF [Bray98] estão a tornar-se familiares para muita gente e estão a ser seguidos e implementados pela indústria.

A Internet foi a grande responsável pelas recentes evoluções registadas na publicação electrónica e no conceito de documento e documento estruturado. Mais à frente, traçaremos o percurso dessa evolução, desde as suas raízes até aos dias de hoje.

Resumindo, pretende-se analisar a tecnologia deste ramo de aplicação e ver de que modo é possível a sua aplicação no âmbito institucional de modo a tornar viável o acesso inteligente a toda a documentação produzida. Por outro lado, interessa-nos também, ver até que ponto a tecnologia associada aos documentos estruturados pode ajudar a melhorar, normalizar e automatizar a publicação electrónica. Aqui, interessa-nos, especialmente, o control de qualidade na vertente associada à correcção de conteúdos.

1.1. A Tese

Face às duas grandes linhas de motivação apresentadas, a existência dum vasto

património documental que é urgente tornar acessível e, a melhoria do processo de publicação electrónica, a tese do autor é a de que a solução para estes dois grandes problemas passa pela utilização da tecnologia associada aos documentos estruturados.

Em muitos casos, a aplicação desta tecnologia resolve apenas parcialmente o problema e noutros, levanta novos problemas. É aqui que surgem os contributos do autor que demonstra conseguir resolver alguns problemas recorrendo à especificação da semântica estática. Contribui também com outras soluções para outros problemas mais pequenos como a normalização de conteúdos e a associação de tipos de dados a conteúdos.

São apresentadas duas vias para a solução da especificação e processamento da semântica estática, a primeira segue uma aproximação via modelos abstractos, a outra, uma aproximação via gramáticas de atributos.

No fim, uma das soluções será escolhida e integrada num sistema que sugere um novo modelo de processamento para documentos estruturados e que explora alguns paradigmas novos neste contexto (adoptam-se para os documentos metodologias utilizadas nas linguagens de programação como consequência duma hipótese levantada pelo autor, da existência dum paralelismo entre o processamento de documentos e o processamento das linguagens de programação), que vão desde a análise da informação até ao seu tratamento. Este novo modelo de processamento, é também enriquecido com uma nova linguagem, definida pelo autor, para a especificação de semântica estática.

A dissertação inclui a apresentação dos passos seguidos na produção do seu próprio texto, uma vez que se adoptaram as soluções defendidas e nela apresentadas, e termina com a apresentação do S4, o sistema de processamento documental que integra e implementa as ideias defendidas ao longo da dissertação.

1.2. Estrutura da Dissertação

Esta dissertação pode ser dividida em duas grandes partes. Na primeira, apresenta-se um estudo da tecnologia existente para trabalhar com documentos estruturados. Na segunda, analisa-se a aplicação da tecnologia a alguns casos reais; identificam-se alguns problemas; e, por fim, propõem-se soluções para a resolução dessas situações.

Assim, a primeira parte começa por um capítulo (Capítulo 3) onde se expõem os conceitos subjacentes aos documentos estruturados: conteúdo e anotação, linguagem de anotação. A seguir surge um dos capítulos basilares da dissertação (Capítulo 4), onde se apresenta um estudo do SGML, o standard para a definição de linguagens de anotação e criação de documentos estruturados. No capítulo seguinte (Capítulo 5), apresenta-se o ciclo de vida dos documentos estruturados, identificando para cada etapa o estado actual de desenvolvimento e utilização.

Esta primeira parte, termina no capítulo onde se começa a discutir a semântica (Capítulo 6). Existe um standard (DSSSL - [Cla96]) para a especificação da semântica dinâmica que é aqui descrito em detalhe. Porém, relativamente à especificação da semântica estática existe o vazio. Excepto alguns invariantes estruturais nativos no SGML, não há nenhuma forma de especificar condições de contexto, ou invariantes, para documentos estruturados. Como aqui já estamos a entrar na parte inovadora desta tese, este assunto passou para o capítulo seguinte (Capítulo 7), onde começa a segunda parte lógica da tese.

A segunda parte inicia-se, portanto, num capítulo onde se discute a especificação da semântica estática. Nessa discussão surgem dois problemas para os quais se apontam soluções: como garantir a normalização de conteúdos; e como associar um tipo de dados ao conteúdo.

Nos dois capítulos seguintes, parte-se para duas soluções de implementação. No primeiro (Capítulo 8), utiliza-se uma abordagem via modelos abstractos e usa-se o ambiente de prototipagem CAMILA para testar a viabilidade da solução. No segundo, utilizam-se gramáticas de atributos como abordagem ao problema e usa-se um ambiente de geração de compiladores para implementar a solução.

Antes dessas duas partes centrais, inclui-se alguma informação sobre formalismos e notação utilizados na dissertação (Capítulo 2).

Por fim, a dissertação termina com um capítulo de conclusões onde são incluídos apontadores para trabalho futuro.

A dissertação possui ainda quatro apêndices. O primeiro descreve a produção da própria dissertação, processo em que se utilizaram a maior parte das metodologias descritas e defendidas ao longo da mesma. O segundo, enumera e descreve sucintamente alguns dos projectos reais onde foram testadas as ideias defendidas nesta dissertação, e que muito contribuíram para enriquecer a visão prática que o autor agora tem do assunto. O terceiro quase que acaba por ser um documento destacável, enumera e descreve um conjunto de regras desenvolvidas ao longo desta

dissertação para a conversão de DTDs em Gramáticas de Atributos. Por fim, inclui-se um quarto apêndice dedicado às novidades que estão a surgir: descrevem-se algumas propostas de standards que poderão revolucionar o mundo dos documentos estruturados e, como efeito lateral, a disponibilização de conteúdos na Internet.

Capítulo 2. Notações e Formalismos utilizados

Ao longo desta tese, faz-se o estudo duma área (processamento de documentos) e identifica-se um problema em particular (semântica estática dos documentos). Para a resolução desse problema seguem-se duas abordagens, cada uma delas suportada por um formalismo, nomeadamente: *CAMILA* (abordagem algébrica no Capítulo 8); e *Gramáticas de Atributos* (abordagem gramatical, no Capítulo 9).

Para facilitar a discussão apresentada nesses capítulos e outras referências ao longo da dissertação, optou-se por introduzir brevemente os referidos formalismos (conceitos básicos e notação) no presente capítulo: *CAMILA* (Secção 2.1); e *Gramáticas de Atributos* (Secção 2.2).

2.1. CAMILA: uma pequena introdução

CAMILA é uma linguagem formal de especificação funcional orientada por modelos algébricos que serve de suporte a um ambiente de prototipagem com o mesmo nome [ABNO97].

Uma especificação CAMILA é um conjunto de componentes de software. Cada um destes é um modelo composto por um tipo abstracto de dados e por definições de funções e de estado [BA95].

A estrutura geral de um modelo é a seguinte:

```
Modelo → MODEL identificador
         DefTipo
         DefFunc
         DefEstado
         ENDMODEL
```

Em que a definição de tipo tem a seguinte forma:

```
DefTipo → TYPE
              (id = Tipo)*
              ENDTYPE
```

Os tipos básicos do CAMILA são apresentados na tabela seguinte, Tabela 2-1.

Tabela 2-1. Tipos Abstractos de Dados e CAMILA

Tipos Abstractos	Notação CAMILA
Conjuntos	X-set
Listas	X-seq
Funções Finitas	X → Y
Relações Binárias	X ↔ Y
União Disjunta	X | Y
tuplos	T = X : A; Y : B
Inteiros	INT
Strings	STR
Tokens	SYM
Universo	ANY

Para além destes tipos, o CAMILA tem mais alguns tipos primitivos que não têm uma correspondência matemática directa mas que são inerentes ao seu ambiente de programação.

Na definição seguinte apresenta-se a forma genérica da definição de uma função em CAMILA.

Definição: uma função em CAMILA

```
DefFunc  → FHeader FPreCond FState FBody
FHeader  → FUNC fid (ParamLst) : typeid
FPreCond → PRE CondExp
FState   → STATE exp
FBody    → RETURNS Exp
```

Por fim, a definição de um estado faz-se como se mostra a seguir.

Definição: um estado em CAMILA

```
DefEstado → STATE sid : typeid
```

Para exemplificar estes conceitos apresenta-se o exemplo seguinte:

Exemplo 2-1. Especificação do modelo Stack

Pretende-se especificar o modelo da stack tradicional.

A declaração dos tipos e do estado são:

```
; ------- SORTS -------------
TYPE
   X = STR;
   Stack = X-list;
ENDTYPE

; ------- STATE -------------
STATE  S : Stack;
```

As funções normais sobre uma stack especificam-se, então, da seguinte maneira:

```
; ------- FUNCTIONS -----------
FUNC push(x:X,s:Stack):Stack
RETURN cons(x,s);

FUNC top(s:Stack):X
PRE s != <>
RETURN head(s);

FUNC pop(s:Stack):Stack
PRE s != <>
RETURN tail(s);
```

Como se pode ver esta versão é puramente funcional, não tem efeitos laterais. As mesmas funções mas agora com os efeitos laterais que provocam alterações no estado, especificam-se da seguinte maneira:

```
; ------- EVENTS -------------
FUNC INIT():
STATE S <- <>;

FUNC PUSH(x:X):
STATE S <- push(x,S);

FUNC POP():X
PRE  S != <>
```

```
RETURN head(S)
STATE S <- tail(S);

FUNC TOP():X
PRE  S != <>
RETURN head(S);
```

A cada tipo CAMILA está associada uma colecção de funções básicas que permitem manusear os valores desse tipo (construir, seleccionar, converter, transformar). Além disso, também estão pré-definidas as conectivas proposicionais e os quantifiadores. Para exemplificar estes operadores, apresentam-se duas tabelas com um resumo das duas colecções de operadores mais usadas ao longo da tese: os operadores para funções finitas; e para sequências. Em cada tabela apresenta-se a sintaxe CAMILA, uma breve descrição informal e a correspondente notação na teoria de *Sets*.

Tabela 2-2. Funções Finitas: X → Y

CAMILA	Descrição
dom(f)	Domínio
ran(f)	Contra-domínio
f[x]	Aplicação
f/s	Restrição de domínio
f\s	Subtracção de domínios

Tabela 2-3. Sequências: X-seq

CAMILA	Descrição
hd(s)	Head
tl(s)	Tail
nth(i,s)	Selecção do iésimo elemento
s^r	Concatenação
< x : s >	Inserção dum elemento à cabeça

CAMILA	Descrição
elems(s)	Conjunto dos elementos
inds(s)	Domínio

As especificações CAMILA podem ser animadas no ambiente de prototipagem designado pelo mesmo nome, CAMILA. Desta maneira, é possível verificar a adequação da especificação ao problema que se está a resolver.

2.2. Gramáticas de Atributos

Uma gramática de atributos é um formalismo muito divulgado e utilizado pela comunidade dos compiladores para especificar a sintaxe e a semântica das linguagens [Hen92].

Introduzidas por Knuth [Knu68], as gramáticas de atributos surgiram como uma extensão das gramáticas independentes de contexto (GIC), permitindo a definição local (sem a utilização de variáveis globais) do significado de cada símbolo num estilo declarativo.

Os símbolos terminais têm atributos intrínsecos (que descrevem a informação léxica a cada um deles associada). Por outro lado, os símbolos não-terminais são associados com atributos genéricos, através dos quais se poderá sintetizar informação semântica (subindo na árvore, das folhas para a raiz), ou herdar informação semântica (descendo na árvore, do topo para as folhas), permitindo referências explícitas a dependências de contexto.

Seja G, uma gramática independente de contexto definida como um tuplo $G=<T,N,S,P>$, onde:

- T representa o conjunto de símbolos terminais (o alfabeto).
- N é o conjunto de símbolos não-terminais.
- S é o símbolo inicial, ou axioma: $S \in N$.
- P é o conjunto de produções, ou regras de derivação, cada uma com a forma:

  $$A \rightarrow \delta,\ A \in N \text{ e } \delta \in (T \cup N)^*$$

  que representaremos neste documento como:

$$X_0 \rightarrow X_1\ X_2\ \ldots\ X_n$$

Para cada regra de derivação $p \in P$ existe uma árvore de sintaxe cuja raíz é X_0, o não-terminal do lado esquerdo, e os seus n descendentes são os X_i com $n \geq i \geq 1$, correspondentes aos símbolos do lado direito. A árvore de sintaxe abstracta respectiva é aquela na qual se retiram os nodos que correspondem a palavras reservadas, só ficando os que representam símbolos com carga semântica.

Dada uma frase f de L_G, a linguagem gerada pela gramática G, designa-se por *Árvore de Sintaxe (AS)*, ou *Árvore de Derivação* , a árvore cuja raíz é o axioma da gramática S, e cuja fronteira (as folhas) é composta pelos símbolos terminais que, uma vez concatenados da esquerda para a direita, formam a frase inicial. A *Árvore de Sintaxe Abstracta (ASA)* obtém-se colando as respectivas sub-árvores abstractas correspondentes a cada uma das regras de derivação $p \in P$ seguidas para derivar a frase a partir do axioma S.

Uma gramática de atributos (GA) é um tuplo $GA=<G,A,R,C>$, onde:

- G é uma GIC que segue a definição dada acima.
- A é a união dos $A(X)$ para cada $X \in (T \cup N)$, e representa o conjunto de todos os atributos (cada um com um nome e um tipo); os atributos dos símbolos terminais chamam-se *intrínsecos* e o seu valor não precisa de ser calculado, provém da análise léxica; para cada não-terminal X, o conjunto dos respectivos atributos $A(X)$ divide-se em dois subconjuntos: os atributos herdados $AH(X)$, e os atributos sintetizados $AS(X)$.
- R é a união dos R_p, o conjunto das regras de cálculo dos valores dos atributos para cada produção $p \in P$.
- C é a união dos C_p, o conjunto das condições de contexto para cada produção $p \in P$.

Seja p uma produção duma GIC ($p \in P$), R_p o respectivo conjunto de regras de cálculo, e C_p o respectivo conjunto de condições de contexto. Neste contexto:

- $X_i.a$, $i \geq 0$, representa o atributo a associado ao símbolo X que ocorre na posição i da produção p.
- uma regra de cálculo é uma expressão da forma $X_i.a = fun(\ ..., X_j.b, \ ...\ )$ com:

  - $a \in AS(X_0)$ V $a \in AH(X_i)$, i > 0
  - $b \in AH(X_0)$ V $b \in AS(X_j)$, j > 0
  - fun é uma função do tipo V_a (o tipo do atributo a)
- uma condição de contexto é um predicado da forma: $pred($ ..., $X_i.a$, ... $)$, $i \geq 0$

Dada uma frase de L_{GA}, a linguagem gerada pela gramática de atributos *GA*, seja ASA a correspondente Árvore de Sintaxe Abstracta, como definido acima. Originalmente, cada nodo da árvore é etiquetado com o símbolo gramatical correspondente e o identificador da produção aplicada para o derivar.

Suponhamos, agora, que cada nodo é enriquecido com os atributos, herdados ou sintetizados, associados aquele símbolo.

Uma *Árvore de Sintaxe Abstracta Decorada (ASAD)* é a ASA inicial depois de passar pelo processo de cálculo dos atributos, que vai associar aos atributos de cada nodo um valor do tipo apropriado. Para que seja uma ASAD válida é ainda necessário que todas as condições de contexto associadas às produções correspondentes aos nodos da árvore, sejam satisfeitas (cujo valor seja verdadeiro para os valores actuais dos atributos).

2.2.1. Cálculo de Atributos

A gramática de atributos é um formalismo declarativo para especificar linguagens. Porém a gramática de atributos pode também ser usada para a construção (manual ou automática) de processadores para as linguagens especificadas (compiladores, tradutores, etc). Assim como da GIC se retira toda a informação para desenvolver o analisador léxico e o analisador sintático (parser), também das regras de cálculo e condições de contexto se pode extrair informação necessária para implementar a análise semântica.

A análise semântica, implementada segundo este paradigma da tradução dirigida pela semântica, é constituída por duas grandes tarefas aplicadas a cada um dos nodos da ASAD:

- o cálculo do valor das ocorrências dos atributos
- a validação das condições contextuais associadas a cada nodo da DAST

Este cálculo faz-se aplicando as funções descritas explicitamente na gramática de atributos sob a forma de regras de cálculo (conjunto R_p), de igual forma a validação realiza-se avaliando os predicados também explicitamente especificados na forma de condições de contexto (conjunto C_p).

As regras de cálculo, ao definirem o valor de uma ocorrência de atributo em funçaõ dos valores de outras ocorrências de atributos, induz imediatamente dependências entre estes. Estas dependências têm de ser identificadas pois irão influenciar a ordem de cálculo, na medida em que devem ser respeitadas para que uma função seja sempre invocada com os argumentos instanciados com os valores correctos. A determinação da ordem de cálculo é uma tarefa complexa (cresce exponencialmente em tempo e espaço relativamente ao número de atributos) que é executada pelo sistema gerador do compilador. Na prática, o problema foi contornado introduzindo a noção de classes gramaticais para as quais se desenvolveram algoritmos de ordenação mais eficientes [Hen92].

Além disso, dada a quantidade de ocorrências de atributos que podem surgir numa ASAD (muitos deles tomando valores em tipos estruturados), o processo de cálculo consome grande parte do tempo total da compilação. Por causa deste factor temporal, surgiu o conceito de *cálculo incremental dos atributos* no contexto de compiladores que são activados em simultâneo com editores estruturados. Nesses ambientes, o compilador tem de ser reactivado (para reanalisar o texto) cada vez que é feita uma alteração a esse texto-fonte. O cálculo incremental (conceito também existente nas "folhas de cálculo") tem por objectivo a minimização do esforço dispendido na análise semântica para que o processo de edição/compilação seja mais rápido. A ideia é manter-se, em tempo real, a informação sobre as dependências entre atributos de modo a que, após uma alteração qualquer na ASA, se identifiquem os atributos associados aos símbolos que foram alterados e aqueles que deles dependem, e se calculem apenas esses valores, sendo conservados todos os restantes.

Como foi dito, para desenvolver parte do trabalho apresentado nesta tese (Capítulo 9), recorreu-se a este formalismo. Para implementar os protótipos, recorreu-se a um ambiente de geração de compiladores incrementais baseado em Gramáticas de Atributos: o *Synthesizer Generator* [RT89a, RT89b, Ram93], que se apresenta na secção seguinte.

2.2.2. Synthesizer Generator (SGen)

Após ter sido introduzido o conceito de Gramáticas de Atributos, apresenta-se agora o *Synthesizer Generator (SGen)* como uma ferramenta que permite gerar automaticamente editores estruturados associados a compiladores incrementais.

O SGen é uma ferramenta que tem por objectivo implementar editores dirigidos pela sintaxe de uma dada linguagem. Estes são gerados a partir de uma especificação formal escrita na linguagem do SGen, a SSL (*"Synthesizer Specification Language"*).

A SSL é uma linguagem funcional que foi pensada e concebida para a especificação de acções semânticas num compilador. Muitos dos ingredientes necessários para o efeito são suportados nativamente na linguagem: manipulação da árvore de derivação, tipos de dados estruturados pensados para suportar tarefas comuns num compilador como por exemplo a gestão duma tabela de símbolos, etc.

O SGen foi desenvolvido na Universidade de Cornell, Estados Unidos, por Thomas Reps e Tim Teitelbaum [RT89a, RT89b]. Surgiu como o sucessor do *"Cornell Program Synthesizer"* [TR81], que era basicamente um editor de PL/I com um compilador e um interpretador incremental.

Este sistema começou a ser desenvolvido em 1981, e devido ao sucesso que alcançou, em 1990, o projecto termina no meio académico e continua numa empresa constituída para o efeito, GrammaTech, Inc.

Os editores gerados pelo SGen trazem algumas vantagens ao desenvolvimento de programas e outras especificações do género:

- a sua maior potencialidade, é a facilidade de compilação incremental.
- o editor gerado pelo SGen faz a análise léxica, a análise sintática e a análise semântica em simultâneo com a edição, podendo ainda, como efeito lateral, realizar o processamento da informação reconhecida até ao momento.
- o conhecimento da sintaxe da linguagem permite que o próprio editor dê um feed-back imediato ao seu utilizador, guiando-o e dispensando-o de conhecer a sintaxe concreta.
- o conhecimento da semântica da linguagem pode ser usado para transformar a informação incrementalmente durante a edição.
- a possibilidade de dotar o editor com o conhecimento estrutural e sintático de uma dada linguagem faz com que estes editores sejam óptimos meios de

normalização de escrita de programas ou documentos. Por exemplo, dotando um editor com o conhecimento estrutural duma carta é possível pôr as pessoas duma empresa a escrever cartas com a mesma estrutura.

- o editor gerado pode ainda ser configurado para ter janelas extra onde são mostradas vistas transformadas da árvore de sintaxe abstracta (*"unpasing rules"*); normalmente estas vistas correspondem às várias gerações de código realizadas pelo compilador.

Um compilador desenvolvido em SGen tem a mesma estrutura dum compilador genérico, tendo portanto os mesmos módulos, só que além destes tem outros relacionados com a interface e a edição estruturada.

A especificação dum editor estruturado em SSL compreende vários módulos: sintaxe abstracta (onde se especifica a sintaxe da linguagem à custa apenas dos elementos estruturais com valor semântico), sintaxe concreta (onde se especifica toda a sintaxe da linguagem; é este módulo que é usado para carregar ficheiros externos no editor), atributos (onde se declaram os atributos e se especificam as suas equações de cálculo) e, por fim, as regras de "unparsing"(onde se especifica de que maneira queremos ver a informação sintetizada na árvore de sintaxe abstracta decorada; para cada transformação pretendida da informação sintetizada iremos ter um destes módulos).

Para exemplos concretos, o leitor poderá consultar as referências fornecidas ou ainda, as duas teses de mestrado desenvolvidas nesta linha de investigação [Lop98, Sou98].

Capítulo 3. Documentação Estruturada

Os documentos são criados com o objectivo de registar informação de modo a ser possível comunicá-la, ou partilhá-la. Para que tenham valor é necessário que sejam fáceis de localizar, fáceis de consumir (interpretar), fáceis de validar e fáceis de reutilizar. A estrutura de um documento é a chave para a informação que nele está contida e pode determinar o seu valor.

As vantagens da explicitação da estrutura, para o aumento de mais-valias da informação baseada em documentos, podem ser analisadas à luz das seguintes perspectivas (traçando um paralelismo com a tecnologia baseada em bases de dados relacionais):

Acesso

: Nas bases de dados relacionais, a rapidez e flexibilidade com que se consegue aceder à informação permite selecionar e ordenar os registos de modo a criar os relatórios necessários num dado momento. Da mesma maneira, a estrutura associada aos documentos permite que elementos destes sejam rapidamente localizados e manipulados. Por exemplo, pode-se seleccionar todos os títulos de capítulos de um livro e construir um índice alfabético.

Validação

: A verificação de que a informação apresentada está completa e de acordo com as regras estruturais é designada por validação. Numa base de dados relacional é sempre possível validar o conteúdo de um campo ou de um registo. Se os documentos forem estruturados correctamente é possível criar algoritmos que executam, de forma aceitável, a validação.

Reutilização

: No contexto de uma base de dados pode-se encarar a reutilização como a possibilidade de gerar diferentes relatórios para o mesmo conjunto de informação. Se, pelo seu lado, os documentos tiverem uma estrutura de algum modo previsível será possível localizar alguns elementos e utilizá-los para

> outros fins. Por exemplo, imagine-se um manual de manutenção em que cada tarefa é sempre identificada com um número e um título descritivo. Deste manual poderiam ser extraídos cartões de tarefa, cada um respeitante a uma tarefa, que seriam entregues aos diferentes mecânicos encarregues de as realizar.

É importante perceber que todas estas vantagens da estruturação só são possíveis se esta fôr rigorosamente mantida. A estrutura dos documentos deve ser monitorada e a sua qualidade controlada ao longo de todo o processo de publicação. Este processo implica um reformular completo dos métodos existentes e uma reconversão por parte das pessoas intervenientes no processo.

3.1. Anotação

Desde há muito tempo, que quem trabalha em publicação de documentos sentiu necessidade de tornar explícitas certos aspectos dos textos que iam ser publicados. Historicamente, o termo inglês *"markup"*, em português *anotação*, *codificação ou etiquetagem*, foi usado para descrever as anotações ou outras marcas que eram colocadas nos textos para instruir os compositores e tipógrafos de como estes deviam ser impressos ou compostos. Por exemplo, palavras sublinhadas com linha ondulada deveriam ser impressas em *"boldface"* (letra mais carregada). Com a automação das tarefas de formatação e impressão o termo começou a ser utilizado num sentido mais amplo, cobrindo todas as espécies de códigos de anotação inseridos em textos electrónicos para dirigir a sua formatação, impressão ou outro tipo de processamento.

Generalizando, podemos definir anotação de um texto como um meio de tornar explícita uma interpretação desse texto. Como exemplo mais banal de anotação podemos olhar para os sinais de pontuação que induzem uma determinada interpretação do texto em que estão inseridos.

A ideia de anotar um documento surge naturalmente quando se pensa num documento específico. Por exemplo, uma carta comercial, é um documento com uma determinada estrutura intrínseca, e quase todas as empresas as escrevem de acordo com essa estrutura (a importância do standard). Ora, para o computador será muito mais fácil processar este tipo de documento se ele estiver *anotado*, i.e., etiquetando o que é o cabeçalho, a assinatura, a data, ... Caso contrário teria que se

percorrer o texto a adivinhar quem é o quê.

Normalmente, os sistemas de processamento de texto requerem que o texto original seja enriquecido com informação adicional. Esta informação adicional, que tem a forma de anotações, serve dois propósitos:

a. Divide o documento nas suas componentes lógicas; e
b. Especifica qual ou quais as funções de processamento que devem ser aplicadas naquele componente.

Em sistemas de DTP (*"DeskTop Publishing"*), que envolvem formatações muito complexas, a anotação é normalmente realizada pelo utilizador, que foi especialmente treinado para desempenhar a tarefa.

Nos sistemas de processamento de texto, as formatações a realizar sobre o texto são mais simples, o que permite que a anotação seja feita pelo utilizador sem um esforço muito consciente. Mas, com o evoluir da tecnologia, mesmo os processadores de texto começam a incluir facilidades de DTP, o que vem trazer uma maior complexidade à tarefa de anotar.

Em conclusão, a tarefa de anotar um texto requer esforço e consome tempo.

Embora o utilizador, na maior parte das situações, não se aperceba, há três passos distintos na tarefa de anotação:

a. Primeiro, há que analisar a estrutura da informação e os atributos que a caracterizam.
b. Depois, é preciso determinar, de memória ou consultando uma norma de estilos, quais as funções de processamento que produzirão o formato/transformação desejado para cada elemento.
c. Por último, há que inserir as anotações no texto.

Distinguem-se dois tipos de anotação: *procedimental e descritiva*. Na primeira, mais orientada a aspectos tipográficos, as anotações vão ter um correspondente impacto físico no aspecto final do documento. A segunda, preocupa-se mais com aspectos lógicos e estruturais do texto, as anotações apenas identificam as várias componentes do documento.

3.1.1. Anotação Procedimental

Observe-se o seguinte exemplo de uma carta, já devidamente anotada:

Exemplo 3-1. Uma carta anotada

```
Comité Organizador da Conferência "Sistemas de Informação pa-
ra o Futuro"
.vspace
Caros Senhores,
Venho por este meio informá-
los dos meios necessários à minha palestra:
.tab 4
.of 4
.vspace
1. um retroprojector
2. um projector com conector VGA para ligação a computa-
dor portátil
.vspace
Obrigado
```

Um sistema de anotação procedimental define qual o processamento a ser realizado em determinados pontos do documento.

No exemplo acima, os comandos **.vspace**, **.tab 4**, e **.of 4**, realizam funções do tipo: deixar uma linha em branco; inserir uma tabulação de quatro posições; e mover a margem esquerda quatro posições.

Como é possível observar, este tipo de anotação é muito pouco flexível e dificultará futuras utilizações com objectivos diferentes do de formatação. As linguagens de anotação procedimental são específicas e dependentes duma plataforma, não são facilmente portáveis para outro sistema com um conjunto de funções e comandos diferente. A anotação procedimental foca o formato e aparência, não tem nenhuma relação com com a hierarquia ou identificação de componentes. Assim, é demasiado ambígua para ser utilizada em aplicações que necessitem de ver o texto com uma estrutura associada. Este tipo de anotação pode, também, tornar-se complexo e de leitura difícil para um humano.

Um dos formatadores mais famosos é o TeX, disponível em várias plataformas. Observe o seguinte exemplo de código TeX:

Exemplo 3-2. Texto anotado em TeX

```
\hrule
\vskip 3cm
\centerline{\bf Escrevendo uma tese}
\vskip 6pt
\centerline{\sl por José Carlos}
\vskip .5cm
```

Como se pode ver, a anotação procedimental pode, muito rapidamente, tornar-se complexa. No entanto, permite um control directo do formato visual do texto no output.

Desde que ao utilizador só interesse criar papel a anotação procedimental serve perfeitamente para atingir os seus objectivos. Se, por outro lado, se se pretender dar outras utilizações ao texto este não é o caminho a seguir.

3.1.2. Anotação Descritiva

Ao contrário da abordagem anterior, a anotação descritiva utiliza códigos de anotação que apenas classificam as componentes do documento.

Códigos como <**P>** ou **\end{description}**, apenas identificam partes do documento e indicam asserções do tipo: *"aqui é parágrafo"*, ou *"este é o fim da última lista descritiva iniciada"*.

Exemplo 3-3. Texto anotado de uma carta

```
<CARTA>
<DEST>Comité Organizador da Conferência "Sistemas de Infor-
mação para o Futuro"</DEST>
```

```
<ABERTURA>Caros Senhores,</ABERTURA>
<CORPO><PARA>Venho por este meio informá-
los dos meios necessários à minha palestra:
<LISTA>
<LITEM>. um retroprojector</LITEM>
<LITEM>. um projector com conector VGA para ligação a compu-
tador portátil</LITEM>
</LISTA></PARA>
<PARA>Por agora é tudo. Até dia 25.</PARA></CORPO>
<FECHO>Obrigado</FECHO>
</CARTA>
```

As vantagens deste tipo de anotação são óbvias: o mesmo documento é susceptível de ser tratado, sem nenhuma alteração, por vários processadores diferentes, podendo, cada um destes, aplicar funções de processamento distintas às várias componentes do documento; cada processador, pode também, tratar apenas as partes relevantes para si. Por exemplo, um programa poderá extrair, de um documento, nomes próprios de pessoas e lugares, para inserir numa base de dados, enquanto outro, processando o mesmo documento, apenas imprimirá os nomes próprios num tipo de letra diferente.

3.1.3. Linguagens de Anotação

Como seria de esperar, as várias comunidades de utilizadores procuram standardizar o conjunto de anotações que utilizam. Ao conjunto de anotações, utilizado por uma dada comunidade de utilizadores num determinado contexto, chamaremos *"Linguagem de Anotação"*. Como se poderá ver mais à frente, existem mecanismos formais para a definição destas linguagens.

Uma linguagem de anotação não define apenas quais as anotações que se poderão utilizar num dado contexto, mas também especifica qual a ordem em que essas anotações devem e podem ou não ser utilizadas.

Existem algumas linguagens de anotação de ampla utilização, embora quem as utilize não tenha muita consciência do que está a fazer. O melhor exemplo é uma linguagem que toda a gente utiliza no dia-a-dia sem se aperceber: os sinais de

pontuação. Os sinais de pontuação que se colocam num texto induzem uma dada interpretação desse texto e dividem-no em componentes.

Outros exemplos, são as linguagens presentes nos sistemas de edição de texto: o LaTeX [GMS94], o RTF e o velhinho WordStar; e linguagens, mais actuais, utilizadas na anotação de documentos para a Internet, o HTML e o XML.

Ao tentar classificar cada uma daquelas linguagens em termos de anotação procedimental ou descritiva, constata-se que essa classificação reflecte a evolução da anotação.

3.1.4. Formatação e/ou Estrutura?

Mesmo utilizando uma linguagem de anotação descritiva, há certos cuidados a ter na definição e utilização dessa linguagem que têm a ver com a subjectividade da tarefa de anotar.

Há três abordagens básicas à tarefa de anotar:

1. Anotação orientada ao formato.
2. Anotação orientada à estrutura.
3. Anotação orientada ao conteúdo.

Uma boa linguagem de anotação terá de surgir de um equilíbrio destas três aproximações. Optando por uma em detrimento das outras poderá levar a lacunas que se farão sentir aquando da anotação de um documento.

3.1.4.1. Anotação orientada ao formato

Este é o processo de anotar o documento com preocupações de estilo e aparência visual. Depois de tudo o que se disse sobre independência de plataformas pode paracer contraditório estar agora a falar em formas mas, há elementos cuja natureza intrínseca está definitivamente ligada à forma. Texto carregado ou em itálico é um bom exemplo disso - o autor pode querer colocar o texto nesta forma de modo a realçar essa parte do texto. Outro exemplo, seria a quebra de página - em certos projectos, a paginação não pode ser livre. Outro exemplo ainda, são as tabelas que representam uma boa maneira de apresentar certo tipo de informação mas que como se disse estão muito ligadas à apresentação.

Exemplo 3-4. Anotação orientada ao formato

```
<TITULO alinha="centrado" estilo="carregado" >Isto é um títu-
lo centrado
e com letra carregada</TITULO>
<P>Isto é um parágrafo onde o autor quis colocar algumas
<REALCE>letras realçadas</REALCE>.</P>
```

O perigo deste tipo de anotação é o de não permitir, no futuro, utilizações alternativas do conteúdo do documento. A anotação orientada ao formato descreve o aspecto visual dum elemento, mas não o identifica nem especifica qual a sua função.

3.1.4.2. Anotação orientada à estrutura

A anotação orientada à estrutura baseia-se na hierarquia genérica. Os elementos são definidos pelo seu nome hierárquico.

Este tipo de anotação torna-se útil quando se tratam vários tipos de estrutura diferentes ao mesmo tempo. Ao definir os elementos em termos da sua posição hierárquica, vão ser necessárias menos anotações do que se estivéssemos a identificar os elementos pelo seu conteúdo.

Exemplo 3-5. Anotação orientada à estrutura

```
<SEC1>Isto é uma secção de nível 1.</SEC1>
<SEC2>Isto é uma secção de nível 2.</SEC2>
<P0>Isto é um parágrafo do nível de topo.</P0>
<LISTA1>Isto é um item duma lista de nível 1.</LISTA1>
<LISTA2>ISTO é um item duma lista de nível 2.</LISTA2>
```

O uso abusivo deste tipo de anotação pode levar a situações em que elementos diferentes em termos de conteúdo mas que ocorrem na mesma posição hierárquica sejam anotados com mesma marca. Se, mais tarde, pretendermos ter um

processamento orientado ao conteúdo não vamos conseguir diferenciar os elementos porque a marca que os identifica é a mesma.

3.1.4.3. Anotação orientada ao conteúdo

A anotação orientada ao conteúdo é o coração da anotação genérica. A identificação dos elementos pelo seu conteúdo e propósito, em lugar da sua forma ou posição hierárquica, faz com que se atinja o expoente máximo da flexibilidade no contexto aplicacional.

Na definição de anotações orientadas ao conteúdo, dá-se nomes diferentes a elementos que tipograficamente são idênticos.

Exemplo 3-6. Anotação orientada ao conteúdo

```
<RECEITA>
  <TITULO>Mousse de Chocolate</TITULO>
  <INGREDIENTES>
    <INGREDIENTE>Meia dúzia de ovos</INGREDIENTE>
    <INGREDIENTE>200g chocolate</INGREDIENTE>
    <INGREDIENTE>50g de manteiga</INGREDIENTE>
  </INGREDIENTES>
  <INSTRUÇÔES>
    <INSTRUÇÂO>Separar as gemas das claras...</INSTRUÇÂO>
    ...
  </INSTRUÇÔES>
</RECEITA>
```

Mais uma vez, o uso abusivo pode levar a situações problemáticas. Neste caso, em situações extremas poderemos estar a anotar elementos com marcas diferentes e, mais tarde, nunca tiraremos partido dessas anotações e as anotações pesam no processamento do documento. Documentos com anotação abusiva tornam-se dispendiosos e muitas vezes impossiveis de processar.

3.1.4.4. Uma anotação equilibrada

Mostraram-se três pequenos exemplos que reflectem os três tipos básicos de anotação. Para cada um destes tipos de anotação, identificaram-se alguns problemas que surgiriam se se optasse por levar ao extremo qualquer um deles.

Depois disto, torna-se claro que a solução óbvia é usar os três de forma equilibrada.

Na escrita desta tese utilizou-se uma linguagem de anotação desenvolvida para a indústria da publicação electrónica, chamada *DocBook* [DocBook,WM99]. Nesta linguagem estão representados os três paradigmas:

Formato

EMPH para realçar texto, e **TABLE** para organizar visualmente a informação.

Estrutura

SECT1, **SECT2** e **SECT3** para marcar os vários níveis de secções e subsecções.

Conteúdo

NAME, **AUTHOR**, **PUBDATE**, **COMMAND**, ..., para marcar vários items de informação.

Podemos pois, prever que num projecto real com alguma dimensão, a anotação a utilizar seja resultado da combinação destes três paradigmas: apresentação, estrutura e conteúdo.

3.2. Documentos e Linguagens de Anotação

O HTML é o formato universal para quem produz documentos para o WWW; a maior parte dos autores nem tem a consciência de que há alternativas. Mas, nos últimos tempos, com o aparecimento do XML, a situação mudou. Estão a aparecer formatos alternativos e mais atractivos. Embora o XML suceda ao HTML, ambos são definidos em SGML que precede o HTML e mesmo a própria Internet. O

SGML foi desenvolvido para dar mais flexibilidade aos autores na expressão do significado do que escrevem, e o XML transportou este conceito para a Internet.

3.2.1. Evolução

A ideia de que os documentos estruturados pudessem ser trocados e manipulados se fossem publicados num formato standard e aberto, data dos anos 60 onde os esforços foram iniciados para que isso se tornasse uma realidade. De um lado, a associação norte-americana *Graphic Communications Association* (GCA), criou uma linguagem designada por *GenCode* para desenvolver anotações para os documentos dos seus clientes que utilizavam diferentes aplicações e geravam diferentes formatos. O *GenCode* permitiu à GCA integrar no mesmo conjunto os vários documentos provenientes desses clientes.

Num esforço paralelo, a IBM desenvolveu outra linguagem designada por *Generalized Markup Language* (GML), na tentativa de resolver os seus problemas internos de publicação electrónica. O GML foi desenhado de modo a permitir que o mesmo documento pudesse ser processado para produzir um livro, um relatório, ou uma edição electrónica.

Como os tipos de documento começaram a proliferar, tendo cada um requisitos diferentes e exigindo por isso um conjunto de anotações diferente, a necessidade para publicar e manipular duma forma standard cada tipo de documento também foi crescendo. No princípio dos anos 80, os representantes do *GenCode* e do GML juntaram-se formando um comité do *American National Standards Institute* (ANSI) designado por *Computer Languages for the Processing of Text*. O seu objectivo era normalizar a metodologia de especificação, a definição, e a utilização de anotações em documentos.

A meta-linguagem SGML, *Standardized Generalized Markup Language*, foi lançada publicamente em 1986 como o standard ISO 8879. É uma linguagem desenhada com o objectivo de permitir a definição e utilização de formatos de documentos. É suficientemente formal para permitir validar os documentos, tem estrutura suficiente para permitir a especificação e manuseamento de documentos complexos, e extensível de modo a suportar a gestão de grandes repositórios de informação.

No fim da década de 80, o SGML tinha já penetrado nalgumas instituições de grande dimensão como a indústria aeronáutica e a de maquinaria pesada. No

entanto, foi no laboratório suiço do CERN, que um investigador então a desenvolver a sua aplicação de hipertexto, lhe achou graça e pensou incluí-la na sua aplicação. Com efeito, Tim Berners-Lee, pai do *World Wide Web* (WWW), escolheu um conjunto de anotações, que retirou dum DTD em SGML, e adoptou essas anotações como a linguagem da sua aplicação. Em Nexus, o editor e navegador original do WWW, ele usou essas anotações, folhas de estilo para formatar visualmente as páginas, e aquilo que lançou definitivamente este género de aplicações: *links*.

Em 1993, altura em que apareceu o Mosaic (primeiro *browser* a ter uma grande divulgação e utilização), os utilizadores estavam já a estender ao máximo o HTML, puxando pelos seus limites. Mesmo a última versão do HTML, a 4.0, lançada em Julho de 1997, fornece apenas um conjunto limitado de anotações. E, se pensarmos um pouco, depressa chegaremos à conclusão de que nenhum conjunto fixo de anotações será suficiente para anotar devidamente todos os documentos que circulam na Internet.

Desde 1992, o HTML evoluiu de uma sintaxe *ad-hoc* para uma linguagem de anotação definida em SGML. A ideia era a de fornecer um suporte formal à linguagem e de que as ferramentas da Internet passassem a implementar o HTML como um caso específico do SGML com uma formatação por defeito, i.e., as ferramentas deixariam de ser específicas do HTML e passariam a ser mais genéricas. Desta maneira, qualquer alteração à sintaxe do HTML seria automaticamente propagada a todas as ferramentas bastando para isso alterar a especificação do HTML e dos estilos. Esta era uma ideia avançada para a época, os seus custos eram grandes e, nessa altura, a Internet não estava preparada para uma linguagem de anotação genérica mas sim para um pequeno conjunto de anotações que qualquer pessoa pudesse aprender numa tarde.

A definição do HTML em SGML foi o primeiro passo para aproximar o SGML da Internet e desta forma cativar o interesse da indústria de software.

Estavam, neste momento, presentes os ingredientes que viabilizariam o aparecimento do XML, *eXtensible Markup Language*, por um lado o reconhecimento do SGML como uma boa metodologia para estruturar e representar documentos, do outro, as limitações do conjunto fixo de anotações do HTML.

3.2.2. O Sentido Ecuménico do HTML

Por incrível que possa parecer, o sucesso do HTML encontra-se associado à sua

grande pobreza semântica. Só um contexto muito pobre pode ser utilizado por uma grande comunidade, um contexto mais rico só pode ser partilhado por uma comunidade pequena e específica.

Actualmente, existem várias comunidades Internet identificadas e que partilham necessidades específicas na anotação dos seus documentos. O SGML possibilita a criação do contexto específico para cada uma destas comunidades. São exemplos: a indústria química, a matemática, o comércio electrónico, a medicina e a música.

O HTML só se pode extender através de pós-processamentos específicos e não através de novos elementos adicionados à linguagem.

O problema fundamental é que o HTML não é unilateralmente extensível (há um organismo que centraliza todo o processo - W3C). Numa linguagem de anotação de utilização genérica, a introdução de uma nova anotação é potencialmente ambígua em termos semânticos (A que é que está associada? Qual o seu papel na estrutura?) e em termos de apresentação (principalmente se não houver referência a uma especificação de estilo).

Pelo contrário, o investimento em SGML possibilita três funcionalidades críticas:

Extensibilidade

: o autor pode definir novas anotações, relacioná-las com as existentes na estrutura, e acrescentar-lhe os atributos que desejar.

Estrutura

: o autor pode associar e definir uma estrutura a um tipo de documento.

Validação

: torna-se possível validar o conteúdo do documento relativamente à estrutura.

O próximo capítulo é dedicado ao estudo do SGML (Capítulo 4), e uma vez que o SGML representa a origem das outras linguagens de anotação que serão discutidas ao longo da tese, aquele estudo introduzirá os conceitos básicos que irão ser necessários ao longo da tese.

Capítulo 4. SGML - ISO8879

Depois de alguma resistência, principalmente da parte da indústria de software, o SGML tem-se vindo a impôr gradualmente a nível mundial. Os seus primeiros clientes residem na indústria que produz mais documentação sobre os produtos que fabrica: a aeronáutica (manuais de construção, manutenção e utilização), a farmacêutica, maquinaria pesada, etc. A seguir vieram as bibliotecas digitais, despolatadas por um projecto piloto na universidade americana Virginia Tech, existem um pouco por todo o mundo.

Hoje, o SGML, é uma realidade, no meio académico, na indústria e no sector terciário. A maioria das empresas já estudou ou está a estudar os custos de uma migração para esta tecnologia.

Neste capítulo faremos várias travessias, em diferentes perspectivas, do SGML, tentando dar uma ideia dos níveis de complexidade e de utilização possíveis para o standard.

Assim, iremos discutir o que são documentos SGML em geral (Secção 4.1), quais as entidades de informação que interagem num sistema SGML (Secção 4.2), e por fim, quais os componentes do SGML e qual a utiliadade de cada um deles (Secção 4.3).

4.1. Documentos SGML

No âmbito da anotação genérica, ou descritiva, o termo documento não refere uma entidade física, como um ficheiro ou um conjunto de folhas impressas. Um documento é visto como uma estrutura lógica encarada como uma hierarquia, identificando-se um elemento especial como a raíz de uma árvore de componentes que constituem o conteúdo do documento. Por exemplo, livro pode ser a raíz de um documento que conterá elementos do tipo capítulo que, por sua vez, podem conter elementos do tipo parágrafo ou imagem.

Os elementos distinguem-se uns dos outros por informação extra – anotações ou marcas – que é adicionada ao conteúdo do documento. Assim, um documento contém dois tipos de informação: dados e anotações.

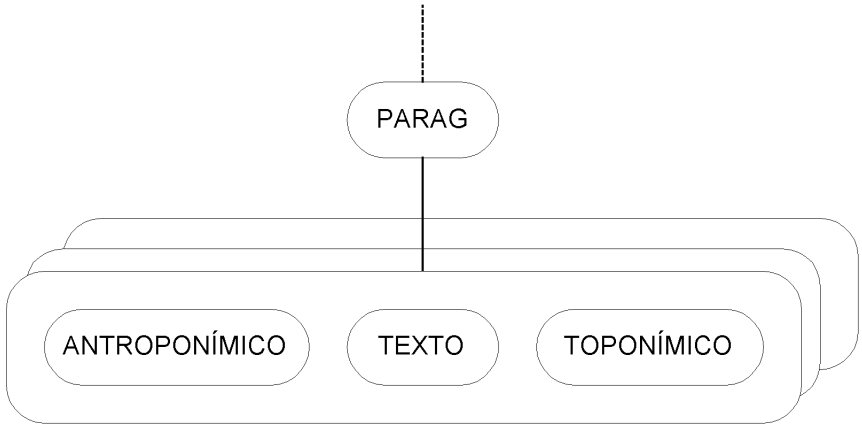
Exemplo 4-1. Documento anotado em SGML

```
<PARAG>O <toponimico va-
lor="PRÚSSIA, Rei da»rei da Prussia</toponimico> anda-
va no exercito.
 Em hua aldeia sentio quatro tiros; hum lhe matou o caval-
lo. Hum guarda julgando o rei morto hia a vingar-
se da atrocidade.
 O rei clamou: --
Estou vivo sem ferida, não haja efusão de sangue. Recolheo-
se a <toponimico valor="VERDUN, Praça de»Praça de
Verdun</toponimico> e ahi ficou sitiado pelos france-
ses. Mas quem não poderia aqui fazer reflexoens?  A
<toponimico valor="PRÚSSIA»Prussia</toponimico> queria is-
to, elle não fazia o negocio deveras.</PARAG>
<PARAG>Entretanto chusmas de franceses expatriados se reco-
lhião a Inglaterra. Aqui se abrirão consignaçoens para reme-
dio destes
miseraveis e nem a diferença de religião nem a politica impe-
dio a generosidade anglicana a esta demonstração de pieda-
de christã. Alguns
ministros protestantes se mostrarão os mais zelosos na carida-
de. A causa dos miseraveis era tocante.</PARAG>
```

Excerto do livro *"Memórias de Inácio Peixoto dos Santos"*.

Como se pode observar, o texto tem os parágrafos devidamente anotados (marca <**PARAG**>) e em cada um deles os nomes próprios relativos a pessoas e lugares estão marcados respectivamente com as anotações <**toponimico**> e <**antroponimico**>.

Figura 4-1. Estrutura do texto anotado



O exemplo apresentado apenas mostra um extracto de uma das partes que constituem um documento SGML. Um documento SGML é composto por três partes distintas:

Prólogo (*"SGML declaration"*)

O prólogo [SGML.Decl] é utilizado para declarações iniciais que vão permitir distinguir as anotações do texto: quais os caracteres delimitadores das marcas; qual o tamanho máximo dos seus identificadores; ...

Normalmente, sempre que não fôr desenvolvido um Prólogo próprio, novo, é assumido um por defeito – o prólogo que se encontra especificado no standard (hoje em dia, insuficiente para a maior parte das aplicações).

DTD (*"Document Type Declaration"*)

O DTD é um conjunto de declarações, escritas na metalinguagem SGML, que no seu conjunto especificam um tipo de documento.

Dois documentos com estrutura diferente (carta, memo, livro, ...), dizem-se de tipos diferentes, ou pertencentes a diferentes classes, e terão por isso DTDs distintos.

Um DTD:

- Define qual a estrutura dum documento.
- Especifica quais as anotações/marcas disponíveis para anotar cada um dos elementos constituintes dos documentos deste tipo.
- Para cada elemento, especifica quais os atributos que lhe estão associados, qual o seu domínio e quais os seus valores por defeito.
- Para cada elemento, define a estrutura do seu conteúdo: que subelementos tem; em que ordem; onde é que pode aparecer texto normal; onde é que podem aparecer dados que não sejam texto.

Texto Anotado (Instância)

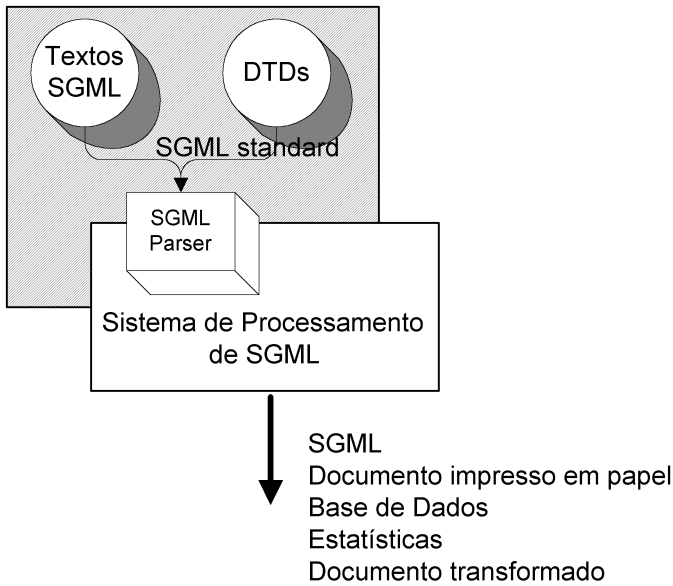
A terceira parte é o documento propriamente dito. Contém a informação (o conteúdo textual), as anotações, e uma referência ao DTD (se este não estiver presente no documento). Um DTD pode ser externo aos documentos, i.e., os documentos concretos (instâncias) podem apenas ter uma referência para o seu DTD que é armazenado noutro ficheiro.

O SGML, com os DTDs veio criar um novo conceito: o de *documento válido*. Um documento é *válido* se a sua estrutura respeitar as regras especificadas no DTD de acordo com o qual é pressuposto que ele tenha sido escrito.

Antes de se pormenorizar mais o SGML (ver Secção 4.3), vai-se contextualizar esta tecnologia (Secção 4.2). O que é que está à volta. O que interage com o SGML e como.

4.2. Arquitectura de um sistema SGML

Figura 4-2. Arquitectura dum sistema SGML



Na figura Figura 4-2, faz-se uma apresentação esquemática dum sistema SGML, com as componentes de informação e de software.

O SGML está por detrás da maior parte dos componentes da arquitectura: nas anotações dos textos; na definição do DTD; e na acção do *parser*. Normalmente, estes componentes são transparentes para o utilizador, estão inseridos no sofware com o qual o utilizador interage, como o editor, o formatador, e a base de dados.

4.2.1. Textos SGML

São ficheiros que contêm textos enriquecidos com anotações definidas num DTD escrito em SGML. Hoje em dia, há editores de SGML bastante poderosos capazes de tornar as anotações invisíveis para o utilizador, validar a estrutura do texto que está a ser inserido e fornecer uma ajuda contextual na escrita desse texto.

O SGML, uma vez que é independente de plataformas de hardware e software, não

especifica nenhuma directiva de comportamento dos editores. Assim, estes nascem apenas das ideias da equipe que os implementa tendo apenas uma coisa que é comum a todos: para se ser um editor SGML tem que se poder importar qualquer texto SGML e, tem que se conseguir exportar qualquer texto criado em SGML.

4.2.2. Um ou mais DTDs

Os DTDs são o elo de ligação dum sistema SGML. Não se pode utilizar SGML sem se pensar num DTD. Não se pode pensar em estruturar um documento sem antes ter pensado e definido uma estrutura para esse tipo de documento.

Um DTD deve ser pensado e definido por um especialista. Este processo deve surgir no início de qualquer projecto editorial, numa fase que se designa por *Análise Documental* (ver Secção 5.1), que é em tudo semelhante à fase de análise na implementação de um sistema de informação.

4.2.3. Um parser

Para assegurar que as anotações num documento estão consistentes com o DTD e sem erros, um sistema SGML tem um programa que reconhece as anotações de um documento SGML, e que lhes aplica um processo de validação. Este programa tem o nome de *parser*.

Um documento SGML deve ser sempre submetido a um parser (passar por um processo de validação) antes de ser exportado para outra palataforma ou antes de ser processado com o objectivo da sua transformação.

Um parser verifica:

- se o DTD está bem definido, de acordo com a sintaxe do SGML;
- se a instância, o texto anotado do documento, está em conformidade com o DTD.

Um editor SGML inclui estas facilidades de validação estrutural, por isso, tem sempre um parser associado. Como se poderá constatar mais à frente, não são apenas os editores que incluem um parser, mas todas as ferramentas que processam textos SGML. No entanto, a maior parte dos parsers de SGML disponíveis têm a

capacidade de funcionar individualmente sem necessidade de estarem dependentes de outras ferramentas.

Existe uma grande variedade de parsers disponíveis, uns comerciais, outros simplesmente livres para quem os quiser utilizar. Curiosamente,e ao contrário de muitas outras tecnologias, são estes últimos, sem encargos comerciais, os mais utilizados, quer pela comunidade de utilizadores SGML, quer pelos produtores de software que os incluem nos editores e formatadores que desenvolvem. Assim temos por ordem de evolução, o ARCSGML [ARCSGML], o SGMLS [SGMLS], e por último o SP [SP].

O ARCSGML é considerado como um dos primeiros parsers a cobrir a maioria da funcionalidade descrita no sandard SGML. O SGMLS é já um parser da nova geração; resultou da reescrita do ARCSGML por James Clark que uns anos depois voltou a reescrever, desta vez o SGMLS, numa implementação orientada a objectos, o SP.

O SP deve ser o parser de SGML mais utilizado, de momento. O seu autor desenvolveu um conjunto de ferramentas que o incluem, que são livres de direitos comerciais, e que são utilizadas por muitas pessoas e empresas no dia-a-dia: nsgmls, sgmlnorm, spam, spent.

4.2.4. Um sistema de processamento

Uma vez que não têm informação específica sobre a sua formatação, os documentos SGML são constituídos apenas pelo conteúdo textual e informação estrutural. Para um utilizador comum, um documento SGML acaba por ser um texto de alguma maneira codificado que ele não entende. Assim, depois de validados pelo parser, os documentos SGML têm que ser processados de modo a serem transformados numa forma mais próxima do utilizador final. A sua estrutura tem que ser traduzida para um conjunto de comandos de um processador de texto, ou de uma base de dados, conforme o objectivo final. Por exemplo, se se quiser uma versão em papel dum documento SGML, para distribuir a um conjunto de leitores, ter-se-á que converter de SGML para RTF, ou Postscript, ou PDF, e usar um formatador para imprimir (no caso do RTF, o MSWord poderia ser usado para criar a versão papel).

O processamento de documentos SGML está fora dos limites do standard. No entanto, a comunidade ligada à publicação electrónica está atenta e em 1996 foi publicado um outro standard ISO que visa normalizar o processamento de

documentos SGML; este novo standard foi designado por DSSSL (*"Document Style and Semantics Specification Language"*) [Cla96,Mul98], e que é analisado na Secção 6.2.

Apesar de ser recente, o DSSSL tem vindo a ser utilizado cada vez mais. Isto deve-se ao facto de, com esta nova peça, se conseguir obter uma abordagem completamente standard para a produção documental, desde a criação do documento até ao output final.

Já existem disponíveis vários sistemas de processamento baseados em DSSSL, uns comerciais e outros livres de direitos. Mais uma vez, o mais divulgado é um sistema sem direitos comerciais, podendo ser utilizado por quem quiser, o Jade [jade].

Neste momento, não há nenhum sistema que implemente uma especificação DSSSL na globalidade das suas potencialidades. O Jade representa a implementação mais conseguida, oferecendo praticamente tudo o respeitante ao processamento de estilos (as primitivas necessárias para formatar graficamente um documento) e a um ou outro detalhe semântico.

O resultado do sistema de processamento pode ser muita coisa, desde o documento traduzido num formato que pode ser carregado num processador de texto até uma lista de nomes colectada ao longo do documento, ou simplesmente um resumo estatístico do conteúdo do documento.

4.3. Componentes dum Documento SGML

Um documento SGML é composto por três grandes e distintas partes: o Prólogo, o DTD e a Instância.

4.3.1. Prólogo

O prólogo é uma parte formal de cada documento SGML. Especifica, por exemplo, quais os caracteres que podem ser utilizados na escrita dos documentos e dentro destes, quais os que serão utilizados como limites das anotações e, muitas outras características de baixo nível como o limite máximo do tamanho dos identificadores das anotações.

Um prólogo

- é, normalmente, comum a todos os documentos SGML num dado ambiente.
- pode ou não fazer parte do documento (no caso da sua ausência será usado um por defeito - o definido no standard).
- fornece detalhes precisos de como o SGML será aplicado ao documento.
- define os caracteres que serão usados para distinguir as anotações do texto (e.g. <, >, />).
- define o conjunto de caracteres (ASCII, EBCDIC ou outro) que vai ser utilizado.

Um utilizador não precisa de conhecer a existência do prólogo para criar e manusear documentos SGML. Uma vez que este define aspectos de muito baixo nível, os sistemas SGML trazem alguns prólogos previamente definidos que permitem aos utilizadores abstrairem-se completamente da sua existência.

Exemplo 4-2. Um Prólogo SGML

```
<!SGML "ISO 8879:1986"

CHARSET

BASESET "ISO 646:1983//CHARSET
         International Reference Version (IRV)//ESC 2/5 4/0"
DESCSET   0   9   UNUSED -
os primeiros 9 caracteres começando em 0
                            não são utilizados-
          9   2   9      -
os próximos, 9 e 10, são mapeados nos
                            caracteres 9 e 10 da ba-
se (BASESET)-
         11   2   UNUSED - o 11 e o 12 não são usados -
         13   1   13     - o 13 é mapeado no 13 -
         14  18   UNUSED -
os próximos 18, começando no 14, não
                            são usados -
```

```
         32  95   32     -
os próximos 95, começando no 32, são mapeados
                            nos caracteres 32-
126 do conjunto base (BASESET) -
        127   1   UNUSED - o caracter 127 não é usado -

CAPACITY PUBLIC "ISO 8879:1986//CAPACITY Reference//EN"
SCOPE    DOCUMENT
SYNTAX   PUBLIC "ISO 8879:1986//SYNTAX Reference//EN"

FEATURES
MINIMIZE
DATATAG  NO
OMITTAG  YES
RANK  NO
SHORTTAG YES
LINK
SIMPLE   NO
IMPLICIT NO
EXPLICIT NO
OTHER
CONCUR NO
SUBDOC NO
FORMAL NO
APPINFO NONE
>
```

No entanto, o prólogo que vem configurado por defeito (ver Exemplo 4-2) está relacionado com as línguas anglo-saxónicas, e compreende apenas metade dos caracteres ASCII. Os países mais ocidentais da Europa, nos quais se inclui Portugal, necessitam do ASCII extendido na escrita dos seus textos. Portanto, na instalação dum sistema SGML deve ter-se o cuidado de instalar um prólogo que inclua os caracteres do ISO LATIN1 como o prólogo por defeito, ou em casos mais complicados incluir o Unicode [Unicode].

4.3.2. DTD

A anotação de um elemento é composta por duas marcas, uma anotação de início do elemento e uma anotação de fim. Estas anotações irão descrever as qualidades características do elemento. Uma daquelas características é o identificador genérico, 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.

O conjunto de anotações num documento, descrevem a sua estrutura. 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 definem todas as estruturas permitidas num determinado tipo de documento.

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

Exemplo 4-3. DTD para uma carta

```
<!- 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 anotadas e ainda, fazer a validação estrutural desses documentos.Contudo, está ainda um pouco distante dum DTD real que contém normalmente maior riqueza informativa.

Formalmente, podemos dizer que um DTD é composto por um conjunto de declarações. Existem quatro tipos de declarações: elementos, atributos, entidades, e instruções de processamento.

4.3.2.1. Elementos

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

```
<!ELEMENT  identificador  - - (exp-conteúdo)  excepção>
```

identificador

: identificador do elemento

expressão de conteúdo

: definição do conteúdo do elemento expressa numa linguagem de expressões regulares que obedece a uma álgebra [CMAlgebra]. A expressão regular que

define o conteúdo do elemento especifica que subelementos podem aparecer, em que ordem e em que número.

excepção

a excepção é opcional e permite modificar a definição do conteúdo do elemento, permitindo elementos opcionais não mencionados na expressão de conteúdo e/ou excluindo outros elementos opcionais pernitidos pela expressão de conteúdo.

4.3.2.1.1. Álgebra do Conteúdo

Nas expressões regulares que podem aparecer a definir o conteúdo dos elementos há dois tipos de operadores: operadores de sequência, ou conexão; e operadores de ocorrência.

4.3.2.1.1.1. Operadores de Conexão

São normalmente colocados entre dois subelementos e especificam a ordem em que podem ocorrer:

,

Exemplo: **(a, b)** – significa que este elemento tem que ser composto por um elemento *a* e um elemento *b*, e que *a* deve preceder *b*.

Do nosso exemplo anterior:

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

uma carta é obrigatoriamente constituída por um destinatário, uma abertura, um corpo e um fecho, nesta ordem.

|

Exemplo: **(a | b)** – significa que o elemento corrente é composto por um elemento *a* ou um elemento *b*.

&

Exemplo: **(a & b)** – significa que o elemento corrente é composto por um elemento *a* e por um elemento *b* sem ordens de precedência, i.e., desde que os dois estejam presentes, a ordem não importa.

Se na declaração anterior de carta substituíssemos os operadores:

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

a constituição dum documento carta seria a mesma, a ordem dos elementos é poderia ser qualquer, por exemplo, uma carta poderia iniciar-se com o fecho.

Na prática, este operador deve ser evitado porque, como iremos ver mais à frente, apesar de nos facilitar a especificação do tipo de alguns documentos mais complicados, vai-nos levantar problemas de processamento e portabilidade.

4.3.2.1.1.2. Operadores de Ocorrência

São aplicados individualmente a um elemento ou a uma estrutura já especificada.

? (0 ou 1 vez)

Exemplo: **(a?,b)** – o elemento que tem este conteúdo tem que ser constituído opcionalmente por um elemento *a* seguido dum elemento *b*.

* (0 ou mais vezes)

Exemplo: **(a*,b)** – o elemento que tem este conteúdo tem que ser constituído por zero ou mais elementos *a* seguidos dum elemento *b*.

+ (1 ou mais vezes)

Exemplo: **(a+,b)** – o elemento que tem este conteúdo tem que ser constituído por um ou mais elementos *a* seguidos dum elemento *b*.

Resumindo, a álgebra do conteúdo define uma linguagem composta por strings de caracteres e elementos do documento. Vamos representar o conjunto de caracteres por Σ. As expressões que definem o conteúdo dos vários elementos num DTD são muito semelhantes a expressões regulares sobre um alfabeto *V*, que é composto

pelos identificadores de tipo dos elementos do DTD e #PCDATA. Iremos referir-nos aos membros de *V* como símbolos. A linguagem *L*(*E*) definida por uma expressão de conteúdo *E* define-se indutivamente como se segue (ϵ representa a string vazia):

```
L(#PCDATA) = {v1...vn|v1...vn ∈ Σ, n ≥ 0}
L(a) = {a} para a ∈ V - {#PCDATA}
L(F|G) = L(F) U L(G)
L(F,G) = {vw | v ∈ L(F), w ∈ L(G)}
L(F?) = L(F) U {ϵ}
L(F*) = {v1...vn | v1, ..., vn ∈ L(F), n ≥ 0}
L(F+) = {v1...vn | v1, ..., vn ∈ L(F), n ≥ 1}
L(F1&...&Fn) = {v1...vn | vi ∈ L(Fϕ(i)), i=1, ..., n, e
    ϕ é uma permutação de {1, ..., n}
```

Num capítulo final, onde discutiremos a implementação dum sistema baseado em SGML voltaremos a abordar esta álgebra de conteúdos referindo alguns problemas de concepção do standard que fazem com que a implementação dum parser SGML seja extremamente difícil e trabalhosa.

4.3.2.1.2. Excepções

Como já foi referido, as excepções permitem alterar a expressão de conteúdo dum elemento, forçando a inclusão ou a exclusão de certos elementos. Trata-se de mais uma característica relacionada com as limitações da tecnologia informática da altura (1986): editores de texto simples, ausência de apoio contextual. As excepções surgiram para abreviar a escrita dos DTDs, no entanto, a sua presença num DTD torna o parsing deste muito complicado.

Os dois tipos de excepção demonstram-se nos exemplos seguintes.

Exemplo 4-4. Inclusão

Se estivéssemos a desenvolver um DTD para o tipo de documentos MEMO podíamos querer tomar a decisão de ter um elemento NOTA-RODAPE "pendurado"em qualquer um dos subelementos de MEMO. Em vez de adicionarmos o elemento NOTA-RODAPE às expressões de conteúdo de todos os

elementos (o que pode não ser simples para expressões complexas), podemos abreviar esta tarefa ligando o elemento NOTA-RODAPE ao elemento raíz MEMO através de uma inclusão.

A sintaxe de uma inclusão é a seguinte:

```
<!ELEMENT nome minimização (exp-conteúdo) +(inclusão)>
```

Assim, a situação anterior seria especificada da seguinte maneira:

```
<!ELEMENT MEMO - - ((PARA & DE),CORPO,FECHO?) +(NOTA-RODAPE)>
```

Por razões de eficiência e dificuldade de processamento que iremos discutir mais à frente, as inclusões devem ser utilizadas com algum cuidado e, preferencialmente, para elementos que não fazem parte do conteúdo lógico no ponto em que ocorrem. Por exemplo: palavras-chave, entradas dum index, entradas do índice, figuras, tabelas, ...

Exemplo 4-5. Exclusão

Pode surgir uma situação em que um elemento deva ser excluído do conteúdo de um dado elemento. Um exemplo disso é o controle do mecanismo de inclusão no sentido de evitar recursividades indesejáveis. No exemplo anterior (Exemplo 4-4), a NOTA-RODAPE não deverá ocorrer dentro do seu próprio conteúdo. Isto pode ser conseguido através duma exclusão.

A sintaxe genérica para as exclusões é a seguinte:

```
<!ELEMENT nome minimização (exp-conteúdo) -(exclusão)>
```

Na continuação do exemplo anterior, introduzimos as seguintes declarações no DTD:

```
<!ELEMENT NOTA-RODAPE - - (PARAGRAFO+) -(NOTA-RODAPE)>
<!ELEMENT PARAGRAFO   - - (TEXTO | NOTA-RODAPE)+>
<!ELEMENT TEXTO       - - (#PCDATA)>
```

Um elemento só pode ser excluído se fôr opcional, se fôr repetitivo ou, se ocorrer numa alternativa. A exclusão de um elemento obrigatório dá origem a uma situação de erro.

Normalmente , as exclusões servem para limitar a recursividade das inclusões. Para evitar que um documento se torne não-estruturado, as exlusões deverão ser usadas, sempre que possível, no nível mais baixo da árvore documental.

4.3.2.2. 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. A especificação de atributos num DTD pode ser comparada à especificação de parâmetros num programa escrito numa linguagem de programação, ou pode-se ainda traçar um paralelo com a nossa língua: os elementos seriam substantivos e, os atributos, os adjectivos. Assim:

```
Isto é uma casa            ==          <CASA>

Isto é uma casa verde    ==          <CASA  COR="verde»
```

Os atributos devem aparecer sempre na anotação que marca o início do elemento, uma vez que vão qualificar o conteúdo que se segue. Não há limite para o número de atributos que podem estar associados a um elemento.

Especificou-se atrás um DTD para o tipo de documento carta. Suponha-se que agora se pretendia classificar cada uma das cartas escritas de acordo com aquele DTD como pessoais ou públicas. Isso pode ser feito associando ao elemento carta um atributo *tipo* com a possibilidade de ter os valores "pessoal"ou "pública":

```
<!ATTLIST  CARTA       TIPO   (pessoal | pública)   pública>
```

Adicionalmente, os valores dos atributos podem ser mais do que simples *strings*, podem ter uma semântica específica. Por exemplo, um atributo pode ser declarado de modo a que o valor que a ele seja associado seja único. Isto é o que se pode chamar de *identificador chave* e que pode ser usado para referenciar e localizar um elemento numa instância dum documento. Este identificador chave pode ser usado por outros elementos para o referenciarem num contexto hipertextual.

A declaração de atributos em SGML tem a seguinte forma:

```
<!ATTLIST elem-id
            att1-id att1-tipo att1-valor-omissao
            ...
            attn-id attn-tipo attn-valor-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-valor-omissao

É o valor por omissão do atributo **i**.

No quadro seguinte, descrevem-se os tipos de atributo mais relevantes e em uso.

Tabela 4-1. Tipos de Atributo

Tipo	Descrição
CDATA	o valor do atributo será uma string (é o tipo de atributo mais usado)
ENTITY	o valor do atributo deverá ser o identificador duma entidade declarada no DTD
ID	o valor do atributo devrá ser um identificador único em todo o documento

Tipo	Descrição
IDREF	o valor do atributo deverá ser um identificador pertencente a um atributo do tipo ID de outro elemento (implementação das referências)
NOTATION	o valor do atributo deverá ser um identificador duma notação declarada no DTD
NUMBER	o valor do atributo é numérico

Por sua vez, o quadro seguinte descreve os valores por omissão dos atributos:

Tabela 4-2. Valores por omissão

#IMPLIED	o valor poderá não ser instanciado, o atributo é opcional.
#REQUIRED	o utilizador terá obrigatoriamente que instanciar o atributo, este é obrigatório
#FIXED	o valor do atributo tem que aparecer a seguir à palavra-chave (ver Secção 7.1, Exemplo 7-3) significando que se o atributo fôr instanciado terá que ser com um valor igual ao declarado, caso não seja instanciado o sistema assumirá esse valor.
#CURRENT	o valor do atributo é herdado da última instância do mesmo elemento onde o valor do atributo foi instanciado (segue-se a ordem ascendente na árvore de elementos)
#CONREF	o valor é usado para referências externas (cada vez menos utilizado, não irá ser considerado nas regras que se seguem)
valor dum tipo enumerado	um dos valores pertencentes ao tipo enumerado definido para esse atributo.

Para terminar a descrição dos atributos, apresenta-se um exemplo comum a quase todas as publicações, as referências bibliográficas.

Exemplo 4-6. Atributos - tipos e valores

Numa lista de items bibliográficos:

```
<BIB.LISTA>
<BIB.ITEM IDENT="Saramago»
<AUTOR>José Saramago</AUTOR>
<TITULO>Jangada de Pedra</TITULO>
...
```

Ao item bibliográfico (<**BIB.ITEM**) atribui-se um identificador chave ao seu atributo **IDENT**, que pode agora ser usado noutros sítios para referenciar este item.

As referências aos items bibliográficos podem ser feitas usando um elemento específico para o efeito, por exemplo, **BIB.REF**, que terá um atributo, por exemplo, **REFIDENT**, que irá ter um valor previamente atribuído a um atributo **IDENT** dum item bibliográfico:

```
O escritor de muitas obras, das quais se destaca "Janga-
da de Pedra" <BIB.REF REFID="Saramago» ...
```

As respectivas declarações em SGML seriam:

```
<!ATTLIST BIB.ITEM IDENT ID #IMPLIED>

<!ATTLIST BIB.REF REFID IDREF #REQUIRED>
```

4.3.2.3. Entidades

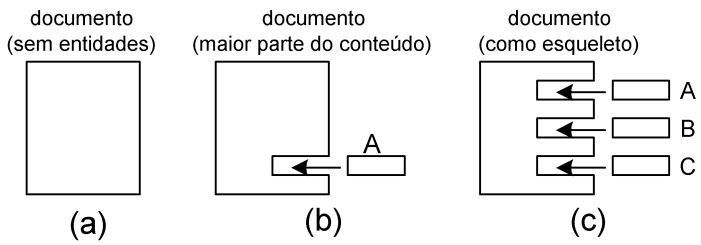
Um documento SGML pode estar espalhado por vários ficheiros do sistema. Facilita-se desta maneira a reutilização de subcomponentes, dentro do próprio documento ou noutros documentos, e permite-se que informação noutro formato seja inserida no documento. No resto desta secção, vai-se descrever o mecanismo do SGML que suporta estas facilidades.

Conceitos

4.3.2.3.1. Conceitos

O SGML tem um mecanismo que permite isolar fisicamente e armazenar separadamente qualquer parte de um documento. Por exemplo, cada capítulo de um livro pode ser gravado em ficheiros distintos, ou, as figuras podem estar guardadas separadamente e podem ser inseridas num dado ponto do documento. Cada uma destas unidades de informação é designada por *entidade* e tem um identificador único associado pelo qual é referenciada. A única excepção é a entidade correspondente ao documento principal que não precisa do identificador pois é normalmente referenciada pelo nome do ficheiro que a contém. Nos casos mais simples, esta será a única entidade relacionada com o documento (o documento SGML está contido num ficheiro - Figura 4-3 a). Noutros casos, o ficheiro principal terá a maior parte do conteúdo do documento, contendo referências a outras entidades que identificam os ficheiros com o conteúdo que preencherá alguns intervalos (Figura 4-3 b). No outro extremo, a entidade/ficheiro principal não é mais do que um esqueleto, usado para posicionar o conteúdo de outras entidades (Figura 4-3 c).

Figura 4-3. Estrutura Física de um documento



Uma entidade é definida numa declaração própria que normalmente aparece no início do DTD. Nesta declaração é atribuído um *nome* à entidade e é-lhe associado um conteúdo ou uma referência para um ficheiro externo onde está esse conteúdo.

As entidades são usadas por referência, i. e., o autor coloca uma referência no texto que identifica univocamente uma entidade. Não há limite para o número de

referências a uma mesma entidade num documento. Mais tarde, quando o documento fôr processado as referências são substituídas pelos respectivos conteúdos.

No texto de uma entidade poderá haver referências a outras entidades. No entanto, não são permitidas referências cíclicas.

Deve-se ter algum cuidado na utilização de entidades e ter alguma atenção na limitação da sua utilização. O seu uso abusivo aumenta a complexidade do tratamento do documento. Há, no entanto, várias situações em que a utilização de entidades deve ser considerada:

- quando a mesma informação é utilizada algumas vezes no documento; a duplicação é sempre susceptível de erros e tem custos temporais.
- quando a informação tem representações diferentes em sistemas incompatíveis.
- quando informação diz respeito a um grande documento que deve ser separado em unidades mais pequenas de modo a facilitar a sua manutenção.
- quando a informação é composta por dados num formato diferente do SGML.

Dos princípios acima enunciados dá para perceber que precisamos de mais de um tipo de entidade de modo a podermos tratar informação SGML e informação que não é SGML, unidades de grandes dimensões e outras de pequenas dimensões:

entidades gerais

: são usadas como um mecanismo de abreviaturas para strings grandes de texto que se irão repetir ao longo de um texto (situação análoga à definição de constantes, ou macros, nas linguagens de programação.

entidades caracter

: representam a maneira de codificar caracteres especiais: caracteres acentuados, letras de outros alfabetos, e alguns símbolos matemáticos.

entidades externas

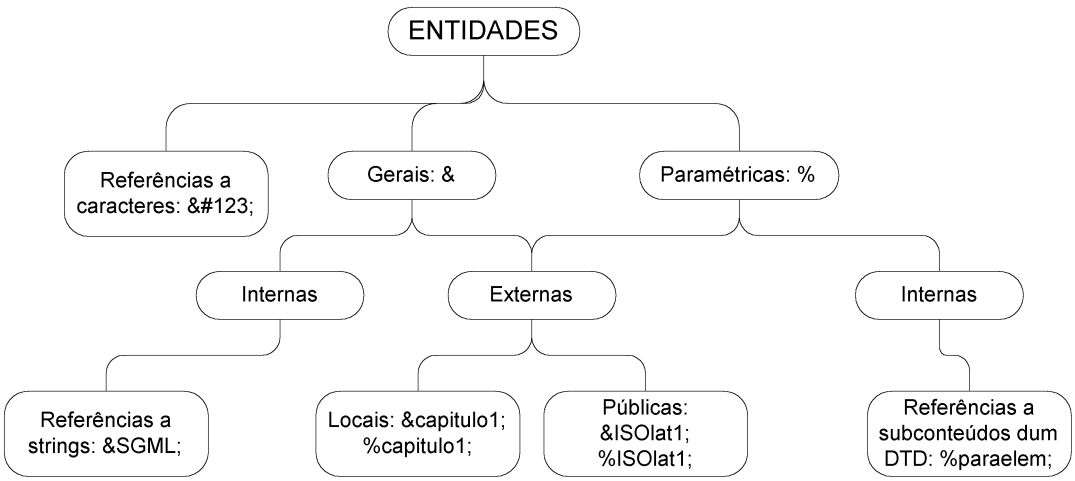
: servem para referenciar ficheiros externos, de modo a ser possível incluí-los nos pontos adequados.

entidades paramétricas

> são usadas como variáveis na escrita de DTDs, para abreviar a sua escrita e tornar as expressões regulares de conteúdo mais simples.

As referências a entidades gerais devem ser colocadas no texto onde se pretender que aquelas sejam expandidas.

Figura 4-4. Os vários tipos de entidades



4.3.2.3.2. Entidades Gerais

Tal como os elementos e os atributos as entidades são definidas no DTD numa declaração própria.

Exemplo 4-7. Declaração de uma entidade geral

Sintaxe da declaração: <**!ENTITY** identificador "conteúdo">

Exemplo:

```
<!ENTITY  SGML  "Standard Generalized Markup Language»
```

Depois, podem ser utilizadas no texto com uma sintaxe específica.

Exemplo 4-8. Referência a uma entidade geral

Sintaxe da referência: **&** identificador **;**

Exemplo:

```
"A migração para uma tecnologia baseada em &SGML; tem cus-
tos que ...
```

Na escrita desta tese foram utilizadas entidades gerais de uma forma peculiar para resolver um problema prático e específico. O DTD usado traz nele incluídas algumas bibliotecas de entidades especiais que mapeiam caracteres matemáticos, gregos, etc. Para a escrita de algumas definições matemáticas estava-se a usar uma biblioteca de letras caligráficas, "ISOmscr.gml". O problema é que nenhum dos "back-ends"utilizados, RTF e HTML, mapeia aqueles caracteres na fonte apropriada e até um dia isso acontecer foi preciso arranjar uma solução de compromisso: mapeou-se cada um dos caracteres num elemento de realce como se pode ver no seguinte exemplo.

Exemplo 4-9. Entidades gerais utilizadas na escrita da tese

```
<!ENTITY ascr "<EMPHASIS>a</EMPHASIS>»
<!ENTITY bscr "<EMPHASIS>b</EMPHASIS>»
<!ENTITY dscr "<EMPHASIS>d</EMPHASIS>»
```

Assim, quando no texto se usa uma daquelas entidades, por exemplo **𝒶**, sempre que o documento é processado a entidade é substituída pelo elemento EMPHASIS com conteúdo igual à letra correspondente. Mais tarde, se os caracteres correspondentes a estas entidades vierem a ter o suporte desejado por parte dos "back-ends", aquelas poderão vir a ser retiradas ou colocadas em comentário.

4.3.2.3.3. Entidades caracter

O conjunto de caracteres definido no prólogo de um documento SGML pode ser muito mais lato do que os caracteres que o teclado disponibiliza. As entidades caracter vêm possibilitar a referência de qualquer caracter definido no prólogo.

O nome destas entidades é o número decimal correspondente ao caracter que se quer referenciar. Por exemplo , se estivermos a trabalhar com os caracteres ASCII a entidade de nome *65* corresponde ao *A*.

Sintaxe da declaração: A sua declaração está intrínseca na declaração dos caracteres no prólogo.

Sintaxe da referência: &# identificador **;**

Exemplo 4-10. Referência a uma entidade caracter

```
... a flecha tinha-lhe trespassado o cora&#132;ão.
```

4.3.2.3.4. Entidades externas

Quando um documento cresce e atinge dimensões consideráveis, a sua edição pode tornar-se penosa e difícil. Numa situação destas, a solução é dividir o documento em unidades mais pequenas, havendo um documento principal a integrar aquelas unidades. Esta tese, por exemplo, encontra-se repartida em vários ficheiros SGML. A política seguida foi a separação por capítulos.

Esta filosofia é implementada em SGML através de entidades externas. Uma entidade externa corresponde a uma das partições do documento original. No documento principal, colocam-se referências às entidades externas no local onde deveriam ser inseridos os respectivos ficheiros SGML.

Exemplo 4-11. Declaração de uma entidade externa

Sintaxe da declaração: <**!ENTITY** identificador [**PUBLIC** identificador-público] [**SYSTEM** path]

Os blocos entre "["e "]"são opcionais mas um deles deverá estar presente. Os dois blocos dizem respeito aos dois métodos possíveis de endereçamento dos ficheiros externos: indirectamente através de identificador público ou, directamente através do nome e local do ficheiro no sistema.

O método indirecto surge para facilitar a gestão dos ficheiros dentro do sistema. Pressupõe a existência de um ficheiro com o nome **catalog**, que faz o emparelhamento entre identificadores públicos e nomes de ficheiros. Um identificador público é uma string que obedece a um conjunto de normas estipuladas num standard Spy97.

Exemplo:

```
<!ENTITY capitulo1 SYSTEM "a:\cap1.sgm»
<!ENTITY capitulo2 PUBLIC -//jcr//Capitulo 2 da tese//PT»
<!ENTITY fig1 PUBLIC -//jcr//Figura 1//PT" "a:\fig1.gif»
```

Aqui estão as três hipóteses para a declaração de uma entidade externa: identificador do sistema, identificador público, e identificador público mais identificador de sistema (este último terá precedência na maior parte das situações.

Como foi referido, a utilização de identificadores públicos é um mecanismo indirecto para o endereçamento de ficheiros que relega para um catálogo a missão de emparelhar os identificadores com os ficheiros do sistema. Apresenta-se a seguir um exemplo de um pequeno catálogo que deveria acompanhar o exemplo anterior.

Exemplo 4-12. Identificadores públicos e o Catálogo

O catálogo referente às declarações anteriores teria o seguinte aspecto:

```
PUBLIC -//jcr//Capitulo 2 da tese//PT" "a:\cap2.sgm"
PUBLIC -//jcr//Figura 1//PT" "a:\fig1.gif"
...
```

A utilidade mais óbvia deste mecanismo é a movimentação de ficheiros. Se pretendêssemos mover os ficheiros no sistema para uma directoria diferente só

teríamos de alterar as respectivas paths no catálogo, caso contrário teríamos que percorrer todos os documentos no nosso sistema e fazer a alateração em todos os que incluíssem as entidades em causa (lembrar que as entidades são reutilizáveis em vários documentos).

As entidades externas são referenciadas como todas as outras que vimos até agora.

Exemplo 4-13. Referência a uma entidade externa

Sintaxe da referência: **&** identificador **;**

Exemplo: ... &capitulo1;...

A seguir apresenta-se um exemplo que mostra como as entidades externas foram utilizadas para modularizar a escrita desta tese seguindo o esquema apresentado na Figura 4-3.

Exemplo 4-14. Entidades externas e escrita modular de documentos

```
<!DOCTYPE BOOK PUBLIC -//Davenport//DTD DocBook V3.0//EN"
[
<!ENTITY capítulo1 SYSTEM "chap-doc-est.sgm" -
Documentação Estruturada->
<!ENTITY capítulo2 SYSTEM "chap-int.sgm" -Introdução->
<!ENTITY capítulo3 SYSTEM "chap-sgml.sgm" -SGML->
<!ENTITY prefacio SYSTEM "prefacio.sgm" -teste de módulos->
]>
<BOOK LANG="pt»
  <BOOKINFO>
    <BOOKBIBLIO>
      <TITLE>Anotação Estrutural de Documentos
...
&capítulo1;
```

```
&capítulo2;
&capítulo3;
...
```

As entidades externas podem ainda ser usadas para incluir num documento objectos externos como imagens, audio ou video.

4.3.2.3.5. Entidades paramétricas

Uma entidade paramétrica é normalmente utilizada na construção do DTD. A sua declaração é semelhante às outras apenas leva o sinal de percentagem, *%*, depois da palavra chave *ENTITY*. Pode ser definida uma entidade geral com o mesmo nome desde que se omita o sinal de percentagem.

Exemplo 4-15. Entidades Paramétricas

```
<!ENTITY % UmaEntidade "(paragrafo | lista)»
<!ENTITY UmaEntidade "Isto é uma entidade.»
```

Para que seja possível distinguir as referências a entidades paramétricas das referências a entidades gerais (uma vez que podem ter o mesmo nome), substitui-se o caracter limitador de início, "&", pelo caracter "%".

4.3.2.4. Instruções de Processamento

Uma instrução de processamento contém informação específica para uma aplicação que irá processar o documento SGML. Ao contrário das outras declarações, uma instrução de processamento não tem regras ou estrutura a seguir, tudo é válido dentro dos limitadores que são respectivamente "<?"e "?>".

O conteúdo de uma instrução de processamento começa normalmente por uma palavra chave significativa para uma das aplicações que irá processar o documento a seguir. Segue-se um espaço que separa a palavra chave do código da instrução de

processamento. É assumido que a sintaxe faz sentido para alguma das aplicações que a seguir irá processar o documento caso contrário, o seu conteúdo será simplesmente ignorado.

Exemplo 4-16. Instruçao de Processamento

```
...
<paragrafo>Seria bom se a página terminasse aqui...
<?TeX \newpage?>
<?HTML <P><HR>?>
</paragrafo>
...
```

Pretende-se que, quer na geração para LaTeX quer na geração para HTML fosse introduzido um separador mais forte no meio do parágrafo. Assim, colocaram-se duas instruções de processamento uma para TeX e outra para HTML. É claro que esta não é a maneira de fazer as coisas respeitando a filosofia do SGML, estamos a adicionar informação específica dependente de plataformas. A maneira correcta de fazer o mesmo seria inserir um elemento vazio que marcaria a quebra de página. De qualquer forma estaríamos a violar parcialmente a filosofia do SGML pois estaríamos a introduzir um elemento com conotações semânticas de forma e não de estrutura.

4.3.3. Instância

A instância é o documento propriamente dito e que é composto por:

- texto - conteúdos.
- anotações - que delimitam os elementos estruturais.
- uma referência ao DTD (caso este não esteja presente no documento).

Usa-se o termo instância porque se trata de uma realização concreta duma classe de documentos que tem a estrutura definida por um DTD.

A título de exemplo apresenta-se uma instância dum documento do tipo CARTA cujo DTD foi definido no início deste capítulo (Exemplo 4-3).

Exemplo 4-17. Instância de CARTA

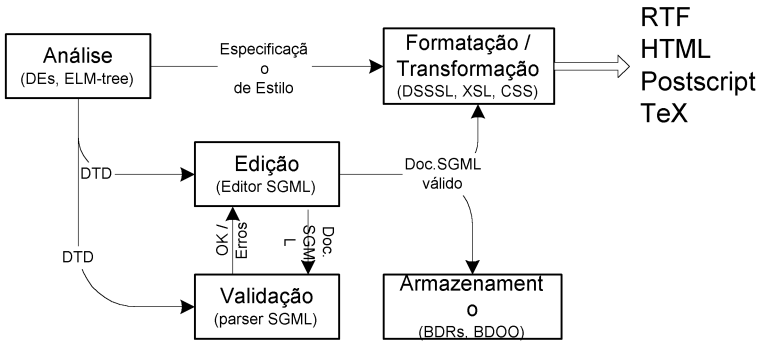
```
<!DOCTYPE CARTA SYSTEM "carta.dtd»
<CARTA>
  <DEST>Meus amigos</DEST>
  <ABERTURA>Caríssimos,</ABERTURA>
  <CORPO>
    <PARA>Na impossibilidade de poder contactá-
los a todos de forma mais personalizada,
      envio esta carta electrónica com os meus vo-
tos de: UM BOM NATAL e BOAS ENTRADAS
      num novo ano que muito promete.</PARA>
  </CORPO>
  <FECHO>Até ao novo ano</FECHO>
</CARTA>
```

Termina aqui a nossa travessia do SGML. No próximo capítulo iremos descrever o ciclo de vida de um documento SGML: que fases, que ferramentas, que outros standards interagem com o SGML, e por fim como obter um produto final de toda esta tecnologia. Não podíamos deixar de fazer aqui uma referência ao arquivo universal de SGML, onde o utilizador poderá encontrar tudo o que desejar desde bibliografia até software, agora mantido pela organização não governamental OASIS responde no endereço internet: *http://www.oasis-open.org/cover*

Capítulo 5. O Ciclo de Desenvolvimento dos Documentos SGML

Os documentos SGML têm um ciclo de desenvolvimento muito bem definido que se pode observar na Figura 5-1.

Figura 5-1. Ciclo de desenvolvimento dos documentos SGML



Eve Maler e Jeanne Andaloussi dedicaram um livro [MA96] a este ciclo de desenvolvimento, com especial relevo na fase de análise. Trata-se de uma publicação que é um marco na bibliografia desta área pois trouxe um pouco de ordem a um universo que estava bastante caótico: detalham-se todos os passos, enumeram-se os formalismos que os suportam e traça-se um perfil da evolução nesta área. O que a seguir se apresenta é um pequeno resumo que inclui esta e outras abordagens que serve para dar uma ideia do que é este ciclo de desenvolvimento.

Como se pode verificar, o ciclo começa numa fase de análise. Esta fase é muito semelhante à fase de análise com que é habitual iniciar o desenvolvimento de um sistema de informação tradicional; neste caso o resultado é a definição de um DTD.

As fases seguintes têm nomes bastante elucidativos: edição/criação, validação, armazenamento e formatação/distribuição. Nas próximas subsecções iremos debruçar-nos sobre cada uma delas.

5.1. Análise Documental

À semelhança de outras áreas da informática, a análise aparece aqui como uma disciplina de modelação de informação, i. e., construção de uma abstracção da realidade mais fácil de manusear que o objecto real. O objectivo desta fase é o de compreender todas as componentes, ou elementos, que constituem uma dada família de documentos, bem como as relações entre eles, de modo a ser possivel desenhar o respectivo DTD com rigor.

Não existe, porém, para um dado contexto, uma única estrutura documental à espera de ser descoberta. Cada um vê aquilo que pretende de acordo com as suas necessidades aplicacionais, como se ilustra no exemplo a seguir.

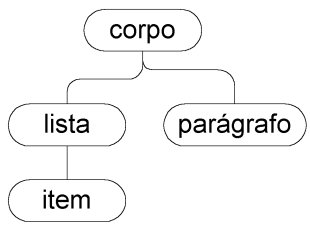
Exemplo 5-1. A Inexistência de uma estrutura única

Veja-se o seguinte texto para o qual se pretende desenhar um DTD:

```
Eis um parágrafo que não contém listas.

1. Primeiro item de uma lista.
2. Segundo item da lista.
```

Uma análise que podemos fazer é a de que documentos deste tipo contêm um corpo com dois elementos - parágrafos e listas - e que a lista ordenada está no mesmo nível hierárquico do parágrafo. Por outras palavras, listas e parágrafos podem ocorrer dentro de secções; no entanto, uma lista não pode ocorrer dentro de um parágrafo e um parágrafo não pode ocorrer dentro de uma lista, resultando uma estrutura lógica do género esquematizado na figura abaixo.



Alternativamente, olhando para o texto do exemplo acima nada indica que o parágrafo está terminado quando a lista começa. Assim, outra possível análise

concluiria que a lista é um subelemento do parágrafo e ocorre somente dentro de parágrafos.

O SGML pode facilmente albergar pequenas variações da estrutura da mesma informação.

Cada pessoa traz uma experiência diferente consigo. Por exemplo, escritores de documentação técnica têm requisitos diferentes, e como tal diferentes visões, dos programadores de bases de dados. Daqui é preciso reter que os resultados da análise documental serão tanto melhores quanto maior e mais representativo fôr o envolvimento de futuros utilizadores da aplicação, na equipe de análise. Desenvolver um DTD deixando de parte um grupo de utilizadores pode resultar numa omissão de elementos que são importantes para esse grupo.

O senso comum é um ingrediente indispensável para assegurar a qualidade de um DTD. Quanto mais pessoas estiverem envolvidas no desenvolvimento do DTD menos chances há de que existam omissões ou grandes erros.

A análise documental é composta pelas seguintes etapas:

- Determinação da área de aplicação.
- Definição de uma estratégia para o DTD.
- Identificação dos utilizadores.
- Escolha de um nome para o DTD.
- Reconhecimento dos elementos lógicos do tipo de documento em causa.
- Escolha entre elementos e atributos.
- Determinação da estrutura hierárquica para o tipo de documento e dos diagramas de estrutura dos conteúdos dos elementos.

5.1.1. Determinação da área de aplicação

Normalmente, quando uma empresa resolve aderir à tecnologia baseada em SGML, pretende reformular todo o seu sistema de suporte à documentação, i. e., pretende que todos os documentos que circulam na empresa estejam integrados, classificados,

e que obedeçam todos à mesma filosofia. Tudo isto faz com que o primeiro passo da análise passe pela identificação dos tipos de documento que circulam e habitam no universo que se pretende remodelar. Eric van Herwijnen [Her94] diz em relação a este ponto: “quando se está a pretender aplicar SGML para sistematizar/formalizar um universo documental, está-se a tentar aplicar ordem ao caos”.

A ideia é agrupar documentos com propriedades semelhantes em classes. Por exemplo, documentos encontrados nas prateleiras de uma biblioteca são *livros*; textos que são enviados de uma pessoa para outra usando algum sistema de correio são *cartas*; documentos produzidos para descrever a utilização de produtos são *manuais*.

5.1.2. Definição de uma estratégia para o DTD

Um DTD desenhado de raíz para documentos que vão ser criados é diferente de um DTD desenhado para suportar documentação já existente (o que acontece quando já existe um património documental).

Portanto, nesta etapa, devemos interrogar-nos sobre os objectivos do DTD que estamos a desenhar, sobre o seu contexto, e sobre a sua compatibilidade com documentos ou DTDs já existentes.

5.1.3. Identificação dos utilizadores

É necessário identificar os potenciais utilizadores para que todos sejam ouvidos. Isto evitará omissões e algum conflito quando mais tarde aqueles forem confrontados com o produto final.

5.1.4. O nome do DTD

Para aqueles que se habituaram a um modo de funcionamento anárquico, o SGML vai causar-lhes alguma frustração pois reduz o grau de liberdade do utilizador; e isto vai reflectir-se a vários níveis. Um deles, é o nome com que se vai baptizar o DTD. Este deve ser elucidativo do tipo de documento que representa e deverá também coincidir com o elemento do topo da hierarquia definida para esse tipo de documento.

5.1.5. Os elementos lógicos do DTD

Um dos resultados da análise é uma lista de elementos, atributos e entidades. Os atributos e as entidades não fazem parte da estrutura hierárquica do documento. Os primeiros identificam propriedades dos elementos. Os últimos refletem a organização física do documento e estão relacionados com a forma de armazenamento e manuseamento do documento. Nesta fase, o objectivo é obter uma descrição da estrutura do documento; interessa, portanto, encontrar e distinguir os elementos que logicamente compõem o documento.

O ponto de partida poderá ser a criação duma lista de todos os elementos que se possam distinguir nos documentos pertencentes ao tipo de documento que se pretende especificar. Veja-se o seguinte memorandum (escrito em SGML):

Exemplo 5-2. Memorandum (SGML)

```
<!DOCTYPE MEMO SYSTEM "memo.dtd»
<MEMO>
<PARA>Camarada Napoleão</PARA>
<DE>Bola de neve</DE>
<CORPO>
<P>Na obra intitulada "A Quinta dos Animais", Geor-
ge Orwell escrevia:
<C>... os porcos tinham muito trabalho, diariamen-
te, com umas coisas
chamadas ficheiros, relatórios, minutas e memos. Es-
tes eram grandes folhas
de papel que tinham de ser cuidadosamente cobertas com escri-
tos, assim que
estivessem cobertas eram queima-
das na fornalha...</C> Será que o SGML
teria ajudado os porcos? Qual a tua opinião?</P></CORPO>
</MEMO>
```

No exemplo acima os elementos lógicos são:

- O elemento **MEMO** que contém todo o memorandum.
- O elemento **PARA** que contém a identificação do destinatário do memo.
- O elemento **DE** que contém a identificação do emissor.
- O elemento **CORPO** que contém a parte textual do memo.
- O elemento **P** que contém o texto dum parágrafo.
- O elemento **C** que contém o texto duma citação.

Durante este processo há questões que devem estar sempre presentes: Preciso mesmo de distinguir este elemento? Que é que quero fazer com ele?

Para que este processo seja o mais completo possível, o analista deve munir-se de vários documentos exemplo pertencentes ao mesmo tipo/classe, e deve rodear-se ou escutar os futuros clientes e utilizadores desses documentos. Isto, na tentativa de cobrir todos os ângulos do problema.

Para o DTD dos memorandus, apesar de não estarem presentes no documento exemplo apresentado, podíamos ainda identificar elementos como **ASSUNTO** e **ASSINATURA**, ou mesmo um elemento mais complexo que visaria registar o percurso do documento dentro da instituição.

Por último, devem ter-se presentes os processamentos a que irão ser submetidos os documentos. Cada processamento pode ter implicações na estrutura.

O conjunto final de elementos deverá suportar todos estes requisitos.

5.1.6. Elemento ou atributo?

Sempre que tentamos criar modelos para objectos do mundo real, somos confrontados com alguns casos marginais, os casos de fronteira. O desenho de DTDs não é diferente. Aqui o caso fronteira surge quando na especificação de um elemento somos levados a hesitar entre elemento e atributo.

Exemplo 5-3. Elemento ou Atributo?

Um bom exemplo de um caso marginal surge na linguagem de descrição de páginas da Internet, o HTML. Em HTML os hyperlinks são definidos por elementos *anchor*

<**A**>. Este elemento pode ter a ele associado uma série de informação: o texto; o URL para onde o browser deverá ir caso o link seja selecionado; um identificador que identifica unicamente o elemento; ... Na definição do HTML (em SGML) o URL é definido como um atributo, de nome **HREF**, do elemento *anchor*. Um link em HTML corrente tem a seguinte forma:

```
...  <A HREF="www.di.uminho.pt/~jcr»Home de Ramalho</A>...
```

Se pensarmos um pouco em termos de estrutura, o URL poderia perfeitamente ser um subelemento do elemento **A**. Ou seja, alternativamente poderíamos definir o elemento em causa com dois subelementos, **LABEL** e **HREF**, de modo a obedecer à seguinte estrutura:

```
... <A><LABEL>Home de Ramalho</LABEL><HREF>www.di.uminho.pt/~jcr</HREF><
```

As duas formas têm a mesma informação, mas a segunda tem mais estrutura. O que poderá ter levado os analistas do HTML a optar pela primeira forma terá sido o processamento semântico que é feito dos links pelos browsers; a forma adoptada facilita parte desse processamento. No entanto, a dúvida "atributo ou elemento?"deve ter persistido durante algum tempo.

Este problema crítico não tem, como boa parte dos problemas de engenharia, uma solução única, definitiva; será resolvido, caso a caso, procurando a melhor solução de compromisso. Contudo, apresentam-se a seguir alguns princípios que poderão ajudar a desambiguar:

- A informação é estrutural? Um objecto que por natureza deve estar obrigatoriamente presente na estrutura (por exemplo, o remetente ou o destinatário de uma carta) é um bom candidato a elemento.
- A informação qualifica o conteúdo ou faz parte de um padrão repetitivo? Uma carta pode ter a ela associada um indicador que identifica o seu tipo como "privada"ou "negócios", não estando relacionado estruturalmente com o elemento carta. Sempre que tivermos informação que pode ser descrita por um valor pertencente a uma lista enumerada, temos um bom candidato a atributo.
- A informação está relacionada com o aspecto visual? Os atributos não devem ser usados para descrever o aspecto visual da informação. Por exemplo, se

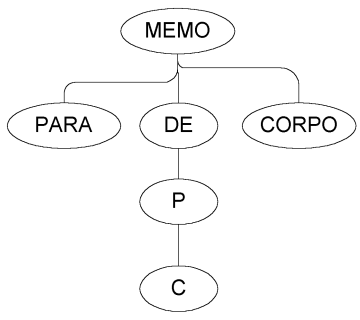
pretendermos realçar certas partes do texto, não devemos fazê-lo recorrendo a atributos, mas sim recorrendo a um novo elemento.

- A informação requer um processamento especial? Esta questão não é muito evidente para quem nunca tenha implementado um processador de SGML, mas é pertinente. O que não é evidente é que quando se opta por utilizar SGML estamos a optar por trabalhar com documentação estruturada em todas as fases do ciclo de vida. O processamento não é excepção e é orientado pela estrutura do documento, pelos elementos. No processamento, os atributos aparecem como algo secundário a que se pode aceder em caso de necessidade para melhor qualificar o processamento, mas o seu conteúdo é visto como um bloco. Portanto não deve colocar-se estrutura no valor de um atributo e qualquer item de informação que requeira um processamento especial deve ser um elemento e não atributo.

5.1.7. Determinação da estrutura hierárquica

Logo que todos os elementos de um dado tipo de documento tenham sido determinados, é necessário especificar as relações entre eles. No exemplo Exemplo 5-2, o elemento **MEMO** é a raiz da estrutura hierárquica, por isso, contém todos os outros. Por sua vez, os elementos **PARA** e **DE** não têm estrutura, apenas texto; o elemento **CORPO** contém uma sequência de parágrafos que por sua vez podem conter texto com alguns elementos pelo meio que identificam citações. Esta estrutura hierárquica pode ser observada na Figura 5-2.

Figura 5-2. Estrutura hierárquica do memorandum



A árvore estrutural do documento dá uma boa maneira de visualizar a sua estrutura; às vezes, torna-se difícil olhar para um texto anotado e tentar perceber qual a sua estrutura. No entanto, a árvore do documento não tem informação nenhuma sobre a frequência de cada um dos elementos: não nos diz quais são opcionais, e não distingue os que podem aparecer com repetições. Para isso teremos que recorrer a outra representação pictórica: os diagramas de estrutura.

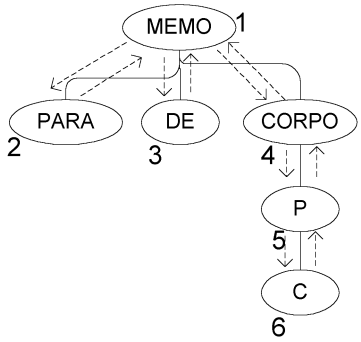
5.1.8. Diagramas de Estrutura

Até agora, como resultado da análise documental, temos um conjunto de elementos e a estrutura hierárquica onde estes se inserem. Daqui até ao que é necessário para escrever um DTD ainda existe alguma distância. Os diagramas de estrutura surgem para colmatar essa diferença. Um diagrama de estrutura é uma representação pictórica de um DTD, sem atributos (estes não têm representação no diagrama).

Para a construção do diagrama de estrutura dum determinado tipo de documento parte-se da árvore esboçada na secção anterior (Secção 5.1.7). Faz-se uma travessia da árvore de cima para baixo e da esquerda para a direita (*depth-first*). A cada nó da árvore vai corresponder um dos oito tipos de diagramas de estrutura. O diagrama de estrutura de cada um dos nós é desenhado olhando para os filhos desse nó e especificando para cada um deles, o seu tipo de frequência.

Para obtermos o diagrama de estrutura do memorandum (Exemplo 5-2), partiríamos da árvore definida (Figura 5-2) e atravessála-íamos da maneira indicada em Figura 5-3; para cada nó desenharíamos o respectivo diagrama de estrutura usando os oito tipos de diagrama que se descrevem a seguir.

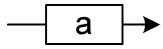
Figura 5-3. Travessia da árvore



Há oito tipos de diagrama de estrutura. Cada um corresponde a uma das estruturas básicas que um dado nó pode ter:

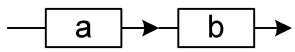
1. Um elemento dentro de uma caixa significa que esse elemento aparece uma vez.

   Figura 5-4. Um elemento

2. Uma caixa seguida de outra, em série, significa que o elemento da primeira caixa precede o segundo e que ambos ocorrem uma vez.

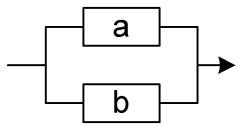
   Figura 5-5. Sequência

3. Duas caixas em paralelo significa que um dos elementos terá de estar presente,

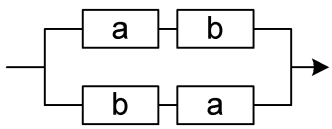
mas apenas um.

Figura 5-6. Alternativa



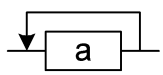
4. Duas caixas em série, em paralelo, com a mesma série invertida, significa que ambos os elementos devem aparecer mas a ordem é livre.

Figura 5-7. Qualquer ordem



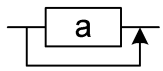
5. Uma caixa com uma seta de feedback significa que o elemento deve aparecer uma ou mais vezes.

Figura 5-8. Uma ou mais vezes



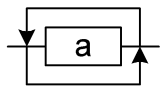
6. Uma caixa com uma seta a fazer-lhe um "bypass"significa que o elemento é opcional.

Figura 5-9. Opcional



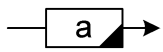
7. Uma caixa com uma seta de feedback e outra de "bypass"significa que o elemento pode aparecer um número indefinido de vezes (zero ou mais vezes).

   Figura 5-10. Zero ou mais vezes



8. Uma caixa, igual ao primeiro diagrama, mas com um triângulo negro no canto inferior direito, é usada se um nodo não tiver subnodos, fôr uma folha.

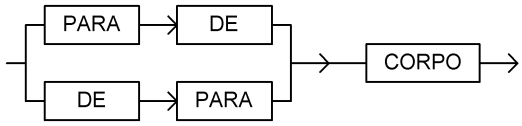
   Figura 5-11. Elemento terminal



Vamos então ver quais os diagramas de estrutura resultantes da travessia da árvore do memorandum:

- Primeiro nodo (marcado com 1 na Figura 5-3) respeitante ao elemento **MEMO**. Tem 3 subnodos, **PARA, DE, CORPO**. Até agora nada foi dito sobre a sua sequência ou frequência. Para tornar o exemplo interessante e elucidativo vamos assumir que **CORPO** vem no fim e que a ordem dos outros dois é irrelevante. Estes pressupostos dariam origem ao seguinte diagrama:

  Figura 5-12. Diagrama de Estrutura do elemento MEMO



- Segundo nodo (marcado com 2): o elemento **PARA** é terminal, o seu conteúdo é apenas texto.

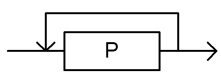
  Figura 5-13. Diagrama de Estrutura do elemento PARA

- Terceiro nodo (marcado com 3): o elemento **DE** também é terminal.

  Figura 5-14. Diagrama de Estrutura do elemento DE

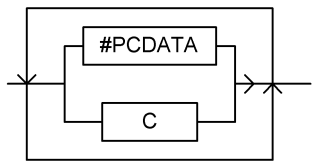
- Quarto nodo (marcado com 4): o elemento **CORPO** é composto por um ou mais parágrafos (**P**).

  Figura 5-15. Diagrama de Estrutura do elemento CORPO



- Quinto nodo (marcado com 5): o elemento **P** tem aquilo que se designa por *conteúdo misto* - essencialmente texto mas podem aparecer algumas citações pelo meio.

Figura 5-16. Diagrama de Estrutura do elemento P



- Sexto nodo (marcado com 6): o elemento **C** é composto por texto.

Figura 5-17. Diagrama de Estrutura do elemento C

Como o leitor atento pode já ter constatado existe quase uma correspondência de um para um entre os tipos de diagramas de estrutura e os operadores de conexão e de ocorrência que se usam para escrever o DTD. Assim, e no caso do memorandum, para escrever o DTD basta escrever a declaração correspondente ao diagrama de estrutura que se determinou para cada elemento. Apresenta-se a seguir o DTD para o memorandum (Exemplo 5-4).

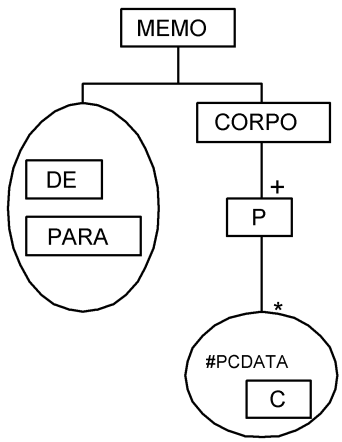
Exemplo 5-4. DTD correpondente aos DEs do Memorandum

```
<!- DTD para o MEMORANDUM - jcr - 99.01.09 ->
<!- Elemento MEMO ->
<!ELEMENT MEMO - - ((PARA & DE) , CORPO)>
<!- Elemento PARA ->
<!ELEMENT PARA - - (#PCDATA)>
<!- Elemento DE ->
<!ELEMENT DE   - - (#PCDATA)>
<!- Elemento CORPO ->
<!ELEMENT CORPO - - (P+)>
```

```
<!- Elemento P ->
<!ELEMENT P - - (#PCDATA | C)*>
<!- Elemento C ->
<!ELEMENT C - - (#PCDATA)>
```

As metodologias de análise baseadas em grafismos são de particular importância em projectos onde o diálogo com interlocutores não informáticos é necessário. Foi apresentada uma metodologia que assenta numa especificação gráfica: árvore hierárquica de componentes e diagramas de estrutura para cada um dos nodos. Para documentos de grande complexidade esta separação pode levar a algumas dificuldades na sua leitura pelo que, mais recentemente, surgiu a ideia de fundir as duas etapas. O formalismo gráfico foi designado por *ELM-tree (Element Lucid Model - tree)* e é introduzido em grande detalhe por Maler e Andaloussi [MA96]. A título de exemplo apresenta-se na figura seguinte a *ELM-tree* para o memorandum.

Figura 5-18. ELM-tree do Memorandum



5.2. Edição de Documentos SGML

Esta fase corresponde à criação das instâncias documentais SGML.

Em princípio, qualquer editor de texto pode servir para escrever um documento SGML. Mas, se quisermos tirar partido, durante a edição, do DTD criado teremos que usar um tipo específico de editor - um editor de SGML.

Um editor de SGML não é mais do que um editor estruturado parametrizável pelo DTD, i.e., é um editor que lendo um DTD vai ser contextualmente sensível a esse DTD. Em primeiro lugar, tais editores ajudam o escritor a marcar o seu texto sem ter de saber de cor a sintaxe de todas as anotações em uso; em segundo lugar, possibilitam a validação estrutural imediata do documento que está a ser editado e outros tipos de pequenas validações que no seu conjunto dão uma grande ajuda ao utilizador na produção de documentos estruturalmente correctos.

Durante o trabalho experimental desta tese, foram vários os editores testados e utilizados em diversos projectos, a saber: FrameMaker+SGML, Emacs+PSGML, Wordperfect8, AuthorEditor, Documentor, XMetal. No Secção A.2 apresenta-se uma análise crítica desses editores utilizados.

No caso da escolha do editor recair num editor SGML, é preciso fazer a sua configuração, trabalho que requer conhecimentos específicos ou, alternativamente, a intervenção do analista que desenvolveu o DTD. A configuração dum editor relativamente a um DTD consiste na compilação deste, isto porque o editor usa os DTDs num formato binário que permite uma maior velocidade em tudo o que diz respeito à validação contextual e este formato obtem-se compilando o DTD com uma ferramenta própria que normalmente acompanha o editor. Depois de compilado, o DTD é adicionado à biblioteca de DTDs do editor ficando disponível para orientar a edição de documentos do tipo nele especificado.

5.3. Validação

A possibilidade de validação é uma das mais-valias que a documentação estruturada trouxe para o universo do processamento e edição documental. Mais especificamente, referimo-nos à validação estrutural, à possibilidade de verificar se a estrutura dum determinado documento respeita um determinado DTD.

Esta tarefa/fase reveste-se de especial importância quando o que está em causa é um sistema baseado em SGML. Um sitema destes gira em torno da validação estrutural. Esta é realizada na edição para validar a introdução do documento, na formatação, na transformação e até na inserção numa base de dados.

Neste caso, esta tarefa é da responsabilidade dum *parser* ou de um *meta-parser* se nos referirmos ao SGML como meta-linguagem. Independentemente da designação que lhe fôr atribuída trata-se de um *parser* diferente, é configurável por uma especificação de estrutura, um DTD, e é este DTD que vai orientar o reconhecimento e validação do documento.

Normalmente, os *parsers* de SGML não aparecem como ferramentas isoladas mas sim integrados noutras aplicações como editores estruturados, formatadores/tranformadores, bases de dados documentais, e outras. Quando a aplicação invoca a acção de *parsing* para validar o documento, o *parser* percorre o documento verificando se este obedece às regras especificadas no DTD e produz dois resultados: um é uma lista de erros que, caso esteja tudo bem, é vazia; o outro é um texto declarativo correspondente à travessia da estrutura de dados interna usada para guardar o documento durante o processo de reconhecimento. Quanto às mensagens de erro, estas variam de acordo com a qualidade do *parser*. Relativamente à linguagem/formato usado para a saída dos dados da travessia, pode haver algumas variações mas há uma assumida como standard: o ESIS – *Element Structure Information Set*.

O ESIS é uma adapatação da notação usada em muitos sistemas funcionais tipo LISP, as expressões-S. A título de exemplo apresenta-se a seguir o ficheiro, em formato ESIS, resultante da aplicação do *parser* ao memorandum do Exemplo 5-2.

Exemplo 5-5. Memorandum em formato ESIS

```
(MEMO
(PARA
-Camarada Napoleão
)PARA
(DE
-Bola de neve
)DE
(CORPO
(P
-Na obra intitulada "A Quinta dos Animais", Geor-
ge Orwell escrevia:\n
(C
```

```
-... os porcos tinham muito trabalho, diariamen-
te, com umas coisas \nchamadas ficheiros, relatórios, minu-
tas e memos. Estes eram grandes folhas\nde papel que ti-
nham de ser cuidadosamente cobertas com escritos, as-
sim que\nestivessem cobertas eram queimadas na fornalha...
)C
-
Será que o SGML\nteria ajudado os porcos? Qual a tua opinião?
)P
)CORPO
)MEMO
```

Cada linha do texto apresentado no exemplo acima tem um significado muito preciso. Descreve-se a seguir os elementos principais da notação utilizada:

(elem-id

: Início do elemento com identificador **elem-id**. Se o elemento tiver atributos, eles já deveriam ter aparecido em linhas com prefixo **A**.

)elem-id

: Fim do elemento cujo identificador é **elem-id**.

-texto

: Conteúdo do último elemento iniciado.

Anome valor

: O próximo elemento a ser iniciado terá um atributo de nome **nome** com o valor **valor**.

?proc-inst

: Instrução de processamento **proc-inst**.

Muitas vezes, o *parser* é invocado não porque seja necessária a validação mas porque é necessário o formato ESIS do documento (o que acontece na formatação, como se poderá ver na Secção 5.5). Como se cabou de ver, o ESIS é um formato em que a informação tem um prefixo que a identifica; torna-se, por isso, um formato ideal para processamento de documentos.

A identificação e a estrutura geradas (que compõem o ESIS) são apenas uma visão diferente mas equivalente da estrutura presente no documento através das anotações e validada pelo DTD.

A construção dum *parser* para SGML é uma tarefa bastante complicada devido a algumas particularidades do SGML. Por exemplo, uma das que levanta mais problemas é o operador **&** que indica a possibilidade combinatória dos seus operandos. Por isso, não é surpresa que até hoje não tenha havido muitas iniciativas para a implementação de *parsers* de SGML.

Porque é a peça central de um sistema baseado em SGML, apresentam-se a seguir os *parsers* existentes:

SP: James Clark's SGML Parser

> Um dos últimos parsers a ser desenvolvido e talvez o mais actual e completo [SP]. Não é bem um parser mas sim uma biblioteca de parsing desenvolvida seguindo uma filosofia orientada a objectos. É utilizada na maior parte dos produtos SGML, comerciais ou não.

SGMLS: James Clark's SGMLS parser

> O antecessor do SP [SGMLS].

ARC-SGML: Charles Goldfarb's Almaden Research Center SGML Parser

> Um dos primeiros parsers a aparecer em público [ARCSGML]. Foi desenvovido enquanto o standard era escrito para validar este último. Foi o antecessor do SGMLS.

ASP-SGML: Jos Warmer's Amsterdam SGML Parser

> Um dos primeiros a ser desenvolvido também. Resultou de uma experiência académica na Universidade de Amsterdão [ASP].

YASP: Pierre Richard's Yorktown Advanced SGML Parser (or: 'Yet Another SGML Parser')

Desenvolvido em 1997 e programado em C++ [YASP]. Não é tão utilizado como os outros.

YAO (Yuan-Ze–Almaden–Oslo project) Parser Materials

YAO é um projecto criado no seio da comunidade SGML [YAO], que visa o desenvolvimento de pacotes de ferramentas para o utilizador, o analista e o programador de SGML com o requisito de que tudo é livre de direitos comerciais. Deste projecto, resultou uma biblioteca de parsing desenvolvida, seguindo uma filosofia orientada a objectos.

Os parsers acima enumerados foram desenvolvidos usando vários paradigmas de programação, embora a maior parte tenha seguido o paradigma da programação orientada a objecto. No entanto, no que diz respeito às técnicas de parsing e compilação propriamente ditas todos seguiram a abordagem tradicional da tradução dirigida pela sintaxe.

No âmbito desta tese, avançou-se com a implementação de um ambiente de programação de documentos SGML (Secção 9.2), onde se seguiu a abordagem da tradução dirigida pela semântica (baseada em gramáticas de atributos), que resolve directamente alguns problemas do parsing de SGML.

5.4. Estilo e especificação da Forma

Tendo acabado de escrever o DTD, esta é a altura ideal para pensar no estilo de apresentação que se quer associar a cada um dos elementos.

É uma tarefa que envolve alguma complexidade e uma vez que o assunto está relacionado com a formatação, deixaremos para a Secção 5.5 a sua descrição.

5.5. Formatação e Transformação

Neste momento, o leitor atento já tem uma boa ideia do que é um documento SGML e algumas perguntas devem estar a perturbá-lo: O que se faz com um

documento SGML? Como se consegue obter uma cópia em papel num dado formato? Como é que é distribuído à população alvo?

Quem opta por uma solução SGML tem de estar consciente de que está a optar por uma solução que separa o conteúdo da formatação. Nas secções anteriores mostramos como é que se pode produzir um conteúdo independente de plataformas de software ou hardware. Agora, vamos ver como é que podemos produzir, a partir desse conteúdo, a forma final desejada.

Duma maneira geral, o que se pretende fazer com um documento SGML é transformá-lo: simplesmente traduzi-lo para RTF, PDF, ou HTML; ou converter a sua estrutura numa estrutura nova dando origem a um novo documento. Neste contexto, a formatação aparece como a preocupação mais imediata de quem produz documentos, mas não é mais que um caso particular da transformação.

Uma solução simples e directa para o problema seria a criação de ferramentas específicas para resolver cada caso particular - uma solução ainda praticável para pequenos casos. Bastaria adoptar, alternativamente, um parser (como vimos na Secção 5.3 alguns destes são disponibilizados como bibliotecas para permitir que os utilizadores os configurem e utilizem à medida das suas necessidades) e acrescentar-lhe o código necessário para realizar a transformação do documento. Ou ainda, adquirir uma ferramenta de transformação que traz um parser embutido e fornece, em termos de programação, uma série de operadores que facilitam esta tarefa) e programá-la para realizar a conversão.

Esta última foi durante algum tempo, e ainda é, uma via aberta para a solução do problema. No entanto, contraria a filosofia que tem vindo a ser seguida desde a criação do documento: a inexistência de dependências em relação a plataformas e a utilização de um formalismo declarativo para especificação da estrutura e anotação do conteúdo. Para conservar estas características era necessário um método normalizado para a especificação da transformação e formatação de um documento SGML.

Estavam então reunidas as condições para o lançamento de um standard para a especificação da transformação/formatação de documentos SGML. Foi assim que em 1996, foi publicado o DSSSL, *Document Style and Semantics Specification Language*, o standard ISO/IEC 10179:1996.

Devido à importância e dificuldade deste tema, criou-se um capítulo específico nesta tese para a sua discussão Capítulo 6.

5.6. Armazenamento

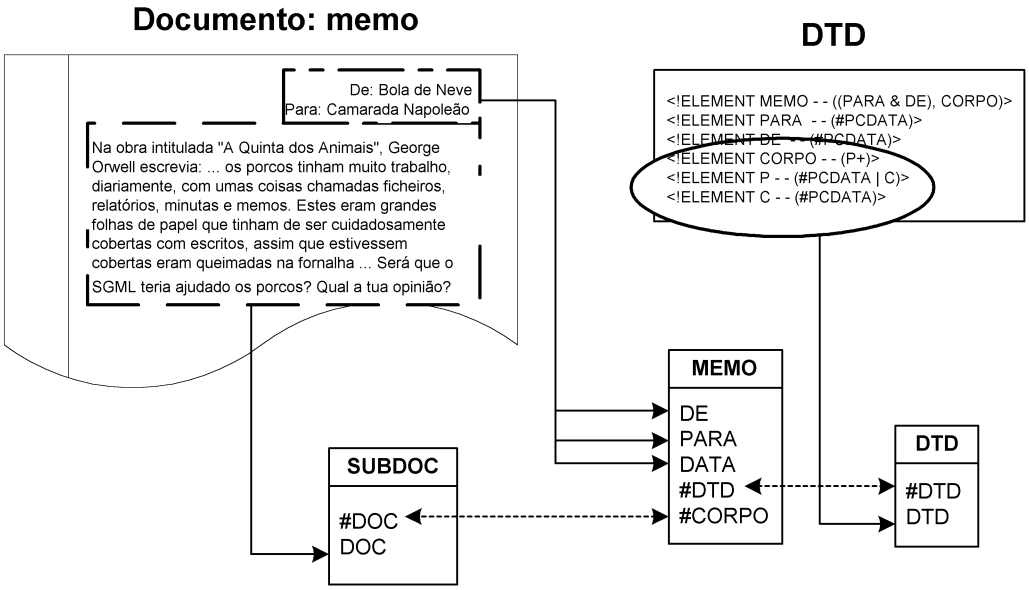
Quem trabalha com documentação estruturada tem por objectivo, tirar partido dessa estrutura para outros fins como: extrair partes; poder realizar procuras inteligentes sobre os documentos; criar índices; extrair partes para construir novos documentos por composição; etc. Para isso é necessário armazenar os documentos num ambiente que permita, posteriormente, realizar aquelas operações sobre eles.

Hoje em dia, é banal a associação das bases de dados com o termo armazenar. No entanto, neste contexto, a solução não é tão simples, há que ter em conta a estrutura da informação e a eficiência do sistema de suporte que se irá utilizar.

Actualmente, podemos dividir o leque de soluções de armazenamento de documentos SGML em três:

1. Armazenar os documentos no sistema de ficheiros do próprio sistema operativo do computador - não é uma solução tão inocente quanto aparenta! Os documentos têm a estrutura visível através das anotações e, se o sistema possuir um conjunto de ferramentas SGML batch que possa ser utilizado para a implementação das procuras e outras operações pretendidas, o problema fica resolvido (com alguma redundância e confusão a nível de ficheiros, uma vez que DTDs, documentos, entidades e ferramentas de software convivem todos no mesmo espaço).
2. Armazenar os documentos numa base de dados relacional - parece uma solução óbvia, mas não é. À primeira vista, podemos ser levados a pensar que basta dividir um documento estruturado nos seus componentes e associar cada um destes a um campo da base de dados. Falso, o texto tem uma ordem linear inerente à sua essência. Por outro lado, não há qualquer relação de ordem entre os campos de uma tabela duma base de dados relacional. Assim, se tentássemos armazenar um documento estruturado numa base de dados relacional distribuindo os seus componentes pelos campos duma ou várias tabelas estaríamos a destruir a ordem linear do texto.

   Figura 5-19. Documentos Estruturados numa BDR

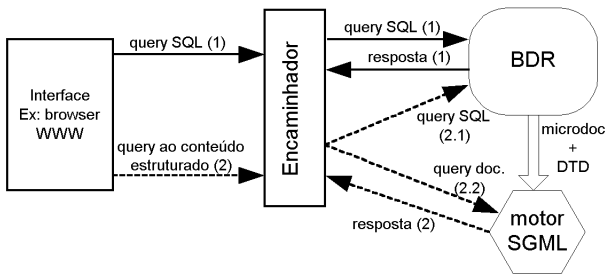


No entanto, e devido aos baixos custos que uma solução baseada no modelo relacional representa, houve várias tentativas nesta área. A mais divulgada, utilizada e até usada como imagem de marca foi a proposta e implementada pela Omnimark [Omn98], designada por *Micro-Document Architecture - MDA*. Esta solução parte dum princípio muito simples: analisa-se a estrutura de um documento e determina-se quais os elementos e subelementos que podem ser separados (que não necessitam da ordem linear do texto), esses elementos são armazenados cada um num campo de uma tabela; depois desta operação resta algum ou alguns subdocumentos para os quais a ordem linear dos seus elementos é importante; estes, são armazenados cada um no seu campo memo que ficará associado a um mini-DTD também armazenado noutra tabela. Desta maneira, os campos de informação que foram separados podem ser utilizados em *queries* normais da base de dados, os outros podem ser geridos como blocos mas se quisermos algo mais inteligente como *queries* ao seu conteúdo estruturado teremos de envolver um motor de processamento de SGML externo, que faça uso, não só desse campo memo, mas também do mini-DTD a ele associado.

Na figura Figura 5-19 apresenta-se uma possível solução para estruturar uma base de dados relacional para documentos do tipo MEMO. Por outro lado, na

figura Figura 5-20, pode-se ver a estrutura funcional que teria de ser montada para que o sistema respondesse a *queries* sobre o conteúdo estruturado dos documentos.

Figura 5-20. Estrutura funcional dum sistema de query para uma BDR documental



3. Armazenar os documentos numa base de dados orientada a objectos - solução cara mas eficaz. O modelo hierárquico de objectos modela directamente qualquer árvore documental portanto nada mais lógico do que utilizar este modelo para armazenar os documentos estruturados. Foi o que pensaram vários construtores de software que desenvolveram vários produtos equivalentes para armazenamento e gestão de documentos estruturados [DuC98], dos quais se destacam: o Astoria, da Chrystal; o Poet, da Poet; e o Infomanager, da Texcel. Deles só podemos dar uma opinião das demonstrações disponíveis, pois o custo das licenças é extremamente elevado e, quando contactados sobre a hipótese de um acordo com a Educação/Universidade, a sua receptividade foi nula.

   Há, no entanto, outros produtos, como o Object Store, que não tendo sido desenvolvidos a pensar no SGML têm sido usados para esse fim, principalmente em projectos de bibliotecas digitais, como a de Alicante - Espanha - "Biblioteca Digital Miguel Cervantes"[DLMC].

De qualquer maneira, a problemática do armazenamento é sempre o último ponto a

ser considerado num projecto. O sistema de ficheiros é um recurso utilizado até ao limite. A decisão de aquisição de um sistema de armazenamento é sempre complicada face aos custos envolvidos.

Com este capítulo, encerramos esta panorâmica geral do que é o mundo dos documentos SGML, como funciona, como é composto, que interacções existem, quais os passos na implementação dum sistema destes.

Nos próximos capítulos iremos entrar na parte inovadora da tese: que lacunas foram detectadas nesta filosofia de trabalho; que soluções o autor propõe para colmatar aquelas deficiências.

Capítulo 6. Documentos e Semântica

Até agora discutimos uma tecnologia que nos permite padronizar a anotação e produção documental e até mesmo validar estruturalmente os documentos que estão a ser produzidos. No entanto, não abordamos ainda dois aspectos fundamentais que são: o conteúdo e o produto final.

Relativamente ao conteúdo, podemos interrogar-nos se não haverá uma forma de garantir certas restrições ou de impôr certas condições de contexto que nos ajudem a detectar e a precaver erros semânticos que, quando acontecem, podem ter consequências graves e embaraçosas, como iremos ver em certos exemplos. Mas, nesta área e enquando esta tese decorria não havia nada que permitisse de alguma maneira realizar esta tarefa. Daí ter sido alvo de um estudo aprofundado que teve como resultados alguns dos contributos mais importantes desta tese. Este assunto foi baptizado de *"Semântica Estática"* e é discutido em detalhe no Capítulo 7.

A questão do produto final é mais crítica que a anterior pois quem produz um documento quer depois vê-lo impresso ou distribuí-lo; se tal não fosse possível de certeza que não adoptaria a tecnologia em causa.

Assim, surgiu um standard que visa normalizar a transformação dos documentos SGML nos vários produtos finais pretendidos pelos utilizadores. Este processo é discutido na Secção 6.2 que foi baptizada com o nome de "Semântica Dinâmica".

Reinventando a Roda? O facto de não existerem metodologias associadas ao SGML que nos permitam especificar a semântica estática, não quer dizer que aquelas tenham de ser desenvolvidas de raíz. Devemos procurar dentro do mesmo domínio se não haverá já soluções desenvolvidas capazes de serem adaptadas a este caso. Assim, na próxima secção, iremos estabelecer uma relação entre o processamento documental e o processamento das linguagens de programação, na tentativa de reaproveitar algumas soluções.

A seguir, apresenta-se uma secção dedicada à semântica dinâmica, mais relevante para o utilizador que quer ver o resultado final logo que possível, e por isso, de alguma maneira, já normalizada há alguns anos.

No fim, na secção dedicada à semântica estática, apresentam-se algumas das inovações mais relevantes deste trabalho, com discussão de casos de estudo.

6.1. Documentos e Programas

Neste momento, podemos traçar um paralelismo entre um documento SGML e um programa escrito numa linguagem de programação. Este paralelismo é mais profundo do que parece; podemos mesmo arriscar e dizer que a evolução que está a sofrer o processamento dos documentos é a mesma já sentida no processamento das linguagens de programação. Assim, surgiu a ideia de estudar este paralelismo e ver se as soluções adoptadas para as linguagens de programação poderiam ser aproveitadas para o processamento dos documentos estruturados, especialmente no campo do tratamento semântico que estava menos desenvolvido.

A tabela seguinte apresenta o resultado da comparação destes dois universos das ciências da computação.

Tabela 6-1. Documentos e Programas

Instância do Documento	Programa
Linguagem de Anotação	Linguagem de Programação
Conjunto de anotações	Vocabulário
DTD	Gramática

Podemos ver o DTD dum documento SGML como uma gramática generativa formal. A partir daqui é fácil estabelecer as relações apresentadas na tabela.

Assim como as gramáticas são o cerne do processamento das linguagens de programação, também os DTDs serão o cerne do processamento dos documentos estruturados. As anotações definidas no DTD correspondem ao vocabulário da linguagem. Uma instância documental escrita de acordo com um DTD é equivalente a um programa escrito numa linguagem de programação.

Um programa teve sempre de ser processado para que se obtivesse o código executável correspondente. O processamento de um programa, ainda hoje designado por compilação, compreendia e compreende as seguintes fases:

- Análise Léxica
- Análise Sintáctica
- Análise Semântica Estática

- Geração de Código, também designada por Processamento da Semântica Dinâmica

No início, o modelo utilizado, e que ainda se utiliza, foi a *Tradução Dirigida pela Sintaxe (TDS)*. Neste modelo, a análise léxica evoluiu e os analisadores léxicos são gerados automaticamente a partir da sua especificação. A análise sintáctica também evoluiu nesse sentido e os analisadores sintácticos são também gerados automaticamente mediante a sua especificação, mas a análise semântica não. Esta é programada pelo autor da linguagem na linguagem hospedeira (linguagem utilizada para implementar os analisadores gerados nas outras fases da compilação).

Se olharmos agora para o universo do SGML, o modelo TDS que se acabou de descrever corresponde ao estado actual de desenvolvimento do modelo de processamento do SGML. O parser de SGML realiza a análise léxica e sintáctica e a análise semântica é deixada para ser programada num ambiente externo.

Em 1968, Donald Knuth introduziu uma nova abordagem: as *Gramáticas de Atributos* [Knu68].

Com *Gramáticas de Atributos* continuamos a poder manter a simplicidade das *Gramáticas Independentes de Contexto* mas agora, com a possibilidade de dar significado às frases à custa de caracterizar os seus símbolos via atributos, e especificar a semântica através de equações sobre esses atributos. Este novo formalismo deu origem a um novo modelo de processamento de linguagens de programação designado por *Tradução Dirigida pela Semântica*.

De alguma forma, o trabalho desenvolvido nesta tese colocou o processamento documental no ponto em que se encontrava o processamento das linguagens de programação no início dos anos 80. Há um formalismo para especificar a semântica, é preciso um motor capaz de o processar. No fim deste capítulo vamos ver as várias hipóteses estudadas e desenvolvidas para permitir a especificação de semântica em documentos estruturados, mas antes iremos ver de onde vem essa necessidade.

6.2. Semântica Dinâmica: o DSSSL

Normalmente, no universo do processamento de linguagens de programação dá-se o nome de "Semântica Dinâmica"à geração de código, devido ao facto de dessa geração resultar o significado operacional (a obter em tempo de execução) do programa. Como nesta secção vamos abordar a geração do documento final,

partindo do documento estruturado original, por analogia com o processo de compilação, nada mais natural do que atribuir-lhe o mesmo nome.

Como o interesse dos utilizadores pelo SGML era crescente, a indústria de software começou a reagir. Estavamos no início dos anos noventa e os editores estruturados baseados em SGML começaram a aparecer no mercado oriundos de esforços empresariais ou académicos.

Como já foi referido várias vezes ao longo deste documento, o SGML especifica apenas estrutura, não fornece qualquer facilidade para a especificação de aparência visual ou formato. Isto criou uma lacuna que começou a ser colmatada da mais óbvia mas pior maneira possível. Cada editor de SGML tinha a sua linguagem própria para associar estilo aos documentos SGML. A portabilidade do SGML estava assim ameaçada, pois os documentos eram apenas parcialmente compatíveis: o mesmo documento SGML podia ser transportado dum editor para outro sem perdas nem necessidade de alterações estruturais ou de conteúdo, mas todo o estilo especificado dentro dum editor era automaticamente perdido quando o documento se manuseava num outro editor diferente.

Foi, pois, na tentativa de normalizar o que faltava do processo que um comité ISO lançou o standard *ISO/IEC 1079:1996* [Cla96], hoje conhecido como *"Document Style Semantics and Specification Language"(DSSSL)* . O DSSSL foi formalmente definido em SGML mas é um pouco diferente das linguagens de anotação que normalmente se definem em SGML; isto deve-se ao facto de a operação de transformação/formatação necessitar de algum processamento. Assim foi preciso dotar o standard de uma linguagem de cálculo. Adoptou-se um subconjunto do standard das linguagens funcionais, o Scheme; este subconjunto forma uma linguagem declarativa e em termos de processamento é livre de efeitos laterais.

Com este novo elemento, já é possível montar uma cadeia de processamento documental completamente standard. Essa cadeia compreende a criação/utilização de três ficheiros e várias acções/processos sobre eles:

- primeiro há que definir a estrutura do documento que se quer criar – utiliza-se o SGML na definição de um DTD, que normalmente é guardado num ficheiro com a extensão ".dtd".
- com um editor, específico de SGML ou não, cria-se a instância do documento, que será depois validada por um parser embutido no editor ou externo a este; normalmente as instâncias dos documentos guardam-se em ficheiros com a

extensão ".sgm".

- uma vez criado o DTD pode e deve-se criar a especificação de estilo em DSSSL para esse DTD, que é guardada num ficheiro com a extensão ".dsl"; para que o sistema produza o resultado desejado, passam-se estes três ficheiros a um motor de formatação que percebe de SGML e DSSSL e que produz o resultado desejado: RTF, TeX, HTML, ou outro.

Fazer um motor de formatação capaz de entender SGML e DSSSL é uma tarefa difícil e penosa. Decorridos já quatro anos desde a publicação do DSSSL apenas três tentativas são dignas de registo: o *SENG* da empresa *"Copernican Solutions"*, o *Hybrick* dos laboratórios "*Fujitsu*", e o *Jade* de James Clark [jade]. O último foi o primeiro a surgir, e aparece dum esforço do editor do DSSSL. Hoje, continua a ser o mais utilizado quer na indústria quer no meio académico, facto a que não é alheia a particularidade de se tratar de um produto livre de direitos comerciais.

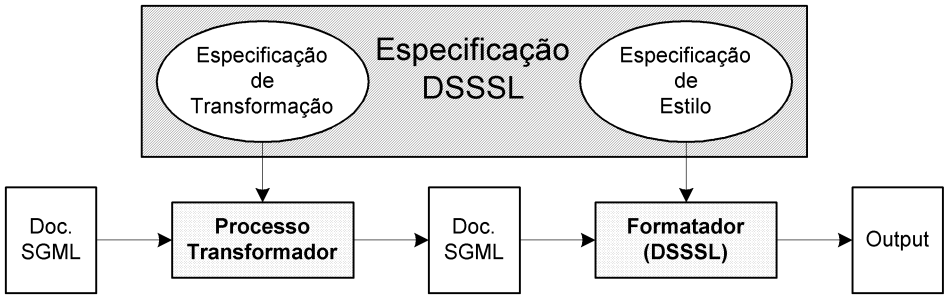
Mais à frente voltaremos a referir-nos a esta ferramenta pois foi e continua a ser utilizada nos trabalhos realizados ao longo desta tese.

6.2.1. Componentes funcionais de especificação

O DSSSL esteve vários anos em desenvolvimento e sofreu várias alterações nesse período. A complexidade que lhe é inerente é bastante alta o que justifica a pouca bibliografia existente apesar dos esforços realizados nesse sentido; além do texto do standard [Cla96], algumas tutoriais bastante superficiais [Ken97, DuC97, Spe98], e duas mais recentes e mais completas [Ger96, Pre97], há apenas aquilo que se baptizou de "DSSSL Documentation Project"que é um esforço global, levado a cabo através da Internet com a colaboração de todos os utilizadores de DSSSL, para a criação de um livro didáctico [Mul98], que sirva quer a principiantes quer a utilizadores experientes.

O modelo conceptual subjacente à linguagem é bastante complexo (Figura 6-1), pelo que vamos analisar a sua estrutura e depois cada uma das suas subcomponentes separadamente.

Figura 6-1. DSSSL - modelo conceptual



O DSSSL é essencialmente uma linguagem de especificação na qual podemos distinguir quatro sublinguagens. Duas principais, para preparar o documento final e formatá-lo:

- uma linguagem para especificar a transformação de um ou mais documentos SGML num ou mais documentos SGML.
- uma linguagem para especificar a aplicação de atributos de formatação a um documento SGML.

E outras duas auxiliares:

- uma linguagem de interrogação, *Standard Document Query Language (SDQL)*, que se pode utilizar para identificar e seleccionar partes de um documento SGML.
- uma linguagem funcional de cálculo de expressões, que é um subconjunto do standard para as linguagens funcionais - *Scheme*, que serve de suporte e de ligação às outras linguagens.

Em termos de implementação, e referindo-nos ao jade, a única sublinguagem totalmente implementada é a de especificação da formatação. Todas as outras foram parcialmente implementadas, e continuam a sê-lo à medida das necessidades. Necessidades estas que têm a ver com a formatação dos documentos, e como é a primeira necessidade dos utilizadores é a que tem de ser servida primeiro para que a tecnologia seja adoptada e tenha sucesso.

6.2.2. Modelo Conceptual

Uma especificação DSSSL compreende duas partes:

- uma especificação de transformação - há situações em que o conteúdo do documento final pretendido difere do original, é um subconjunto ou corresponde a uma reordenação do conteúdo inicial; nestas situações o documento original tem que passar por uma fase de transformação antes de ser formatado.
- uma especificação de estilo - que vai dirigir a formatação.

O transformador actua sobre o documento SGML de origem e transforma-o, de acordo com a especificação, dando origem a um novo documento SGML. Este novo documento é por sua vez passado a um formatador que o vai formatar de acordo com a especificação de estilo dando origem a um documento no formato previamente seleccionado para output (RTF, Postscript, PDF, HTML, etc).

A especificação de estilo controla parcialmente o processo de formatação (definição de margens, escolha do tipo de letra, tamanho da letra). O controle é apenas parcial porque há muitos detalhes que são pré-estabelecidos pela equipe responsável pela implementação do formatador e que podem diferir de uns para os outros. Por exemplo, os algoritmos que gerem a quebra de linha e de página podem ser implementados de várias maneiras.

Estes dois processos, transformação e formatação, podem constituir uma só aplicação ou, poderão ser duas aplicações completamente independentes. Não há qualquer dependência entre eles.

6.2.3. Linguagem de Transformação

Como já foi referido, a linguagem de transformação permite especificar processos de modificação (alteração/eliminação/criação).

Um processo de transformação pode compreender operações como:

- Rearranjo de estruturas - reordenação e/ou agrupamento de estruturas existentes.
- Construção de novos elementos relacionados com outros elementos já existentes - o analista especifica como é que os novos elementos se obtêm a partir dos

existentes - por exemplo, as **footnotes** que vão aparecendo podem ser agrupadas e colocadas no fim do respectivo capítulo.

- Associação de novas características a sequências específicas de conteúdo - o primeiro parágrafo de um capítulo pode ter um início diferente dos outros parágrafos nesse capítulo - estas sequências de conteúdo podem ser identificadas recorrendo à utilização de SDQL.
- Associação de novas características a componentes específicos de conteúdo - por exemplo, associar atributos de formatação a strings do texto que não se encontram anotadas em SGML.

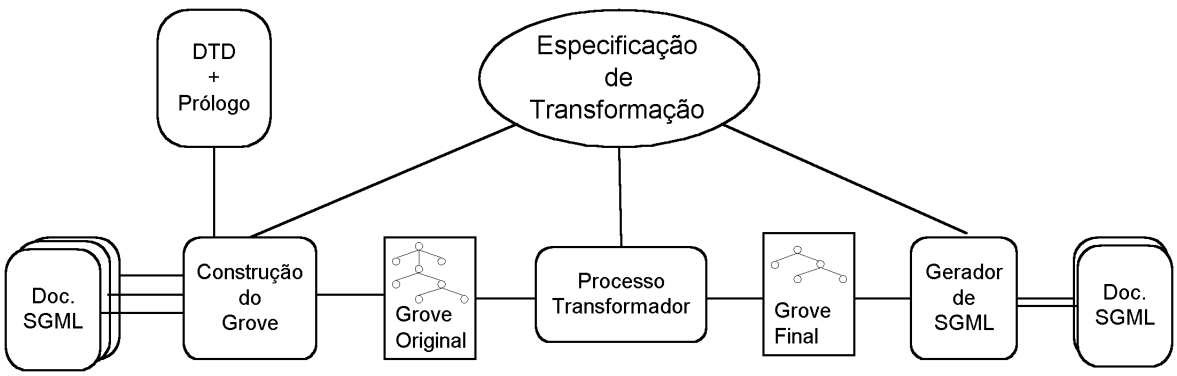
Resumindo, com esta linguagem o utilizador pode especificar modificações que afectam, quer o conteúdo, quer a estrutura do documento SGML original.

Como se disse acima, todas as operações especificadas no processo de transformação são independentes do processo de formatação que se lhe vai seguir.

6.2.4. O Processo de Transformação

O processo de transformação, *SGML Tree Transformation Process (STTP)*, representado na Figura 6-2, compreende as etapas, que se descrevem a seguir: Construção do Grove (Secção 6.2.4.1); Transformação (Secção 6.2.4.2); Geração de SGML (Secção 6.2.4.3).

Figura 6-2. DSSSL - processo de transformação



6.2.4.1. Construção do Grove

O documento SGML é submetido a um parser que vai reconhecer a sua estrutura e que o vai guardar numa estrutura interna especial - o *Grove*. Um *grove* é a representação em grafo da estrutura e conteúdo de um documento; trata-se de um grafo orientado e pesado em que cada nodo é fortemente tipado (cada nodo tem uma etiqueta associada que responde pelo seu tipo: no Exemplo 6-1, temos nodos do tipo *receita*, *titulo*, ...) e relaciona-se de várias maneiras com os seus vizinhos (podemos ver uma relação como o peso associado a um ramo, no Exemplo 6-1 temos as relações: *filho/conteúdo*, *irmão*, *atributos*). Deste modo, a estrutura pode ser atravessada de diferentes formas, de acordo com o fim em vista. Podemos por exemplo, realizar pesquisas contextuais do tipo: "*Dê-me todas as receitas cujo título contem a palavra BOLO*"; ou operações de filtragem do tipo: "*Lista de todos os títulos de receita*".

A estrutura utilizada nas ferramentas mais completas, como o *jade* [jade], é bastante complexa; no entanto, depois de se ter observado cerca de meia centena de equipes a pensar sobre o assunto[1], concluiu-se que a estrutura mínima para representar um documento estruturado, especificada em CAMILA, seria a seguinte:

```
Grove     : Nodo-Seq
Nodo      : identificador * Atributos * Conteúdo
Atributos : Atributo-Seq
Atributo  : identificador * tipo * valor
Conteúdo  : Nodo-Seq
```

Apresenta-se a seguir um exemplo prático de um grove (Exemplo 6-1).

Exemplo 6-1. Grove - estrutura para manipulação de documentos SGML

Eis uma instância dum documento SGML muito simples a partir do qual se irá construir o grove:

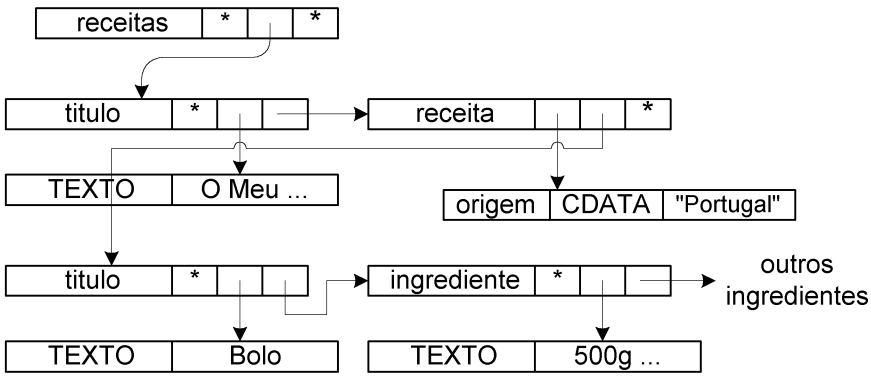
```
<RECEITAS>
  <TITULO> O Meu Livro de Receitas </TITULO>
  <RECEITA ORIGEM="Portugal»
    <TITULO> Bolo </TITULO>
    <INGREDIENTE> 500g de farinha </INGREDIENTE>
    <INGREDIENTE> 200g de açucar </INGREDIENTE>
```

```
    <INGREDIENTE> 300g de manteiga </INGREDIENTE>
  </RECEITA>
</RECEITAS>
```

O qual foi escrito de acordo com o seguinte DTD:

```
<!DOCTYPE RECEITAS [
<!ELEMENT RECEITAS    (TITULO,RECEITA*)>
<!ELEMENT TITULO      (#PCDATA)>
<!ELEMENT RECEITA     (TITULO,INGREDIENTE+)>
<!ELEMENT INGREDIENTE (#PCDATA)>
<!ATTLIST RECEITA
                      ORIGEM CDATA #IMPLIED>
]>
```

Na figura abaixo apresenta-se o grove correspondente ao documento apresentado em cima.



Os nodos com o identificador TEXTO são nodos com conteúdo documental, ao contrário dos outros que contêm informação estrutural.

Podemos também ver as relações acima faladas: na primeira posição do tuplo que representa cada um dos nodos do grafo temos o identificador do tipo de nodo, na segunda posição temos o ramo correspondente à relação *atributos*, na terceira o ramo correspondente à relação *filho*, e na quarta o ramo correspondente à relação *irmão*.

Em groves mais complexos, é normal aparecerem representadas as relações inversas (é apenas uma decisão de implementação e está relacionada com a eficiência de algumas operações.

Para mais informação o leitor pode consultar o texto do standard [Cla96] ou o já citado livro de DSSSL que está a ser escrito com as contribuições de todos através da Internet [Mul98].

6.2.4.2. Transformação

A Transformação propriamente dita é o cerne desta primeira fase sendo realizada por um sistema de tradução que vai reescrevendo a informação de cada nodo de acordo com regras *Condição-Acção*: uma regra é activada (seleccionada) sempre que a respectiva *Condição* sobre o nodo corrente (da travessia que o processo está a realizar) tiver o resultado verdadeiro; então a *Acção* é executada (normalmente corresponde a uma transformação do conteúdo do nodo corrente cujo resultado é enviado para a saída).

A especificação das operações de transformação é utilizada para controlar este processo que parte do grove do documento inicial e o transforma no grove do documento final.

A especificação de transformação é composta por um conjunto de associações. Cada associação é um triplo: uma expressão de *query* (para seleccionar os componentes do documento que irão ser afectados pela transformação) - aqui usa-se a referida SDQL; uma expressão de transformação - aqui usa-se Scheme; e uma expressão de prioridade, que é opcional (não se usa na maior parte dos casos).

Em CAMILA poderíamos definir uma especificação de transformação da seguinte forma:

```
          Espec-Transf : Associação-Set
          Asso-
ciação   : Exp_Selecção * Exp_Transformação *Exp_Prioridade
```

Devido à complexidade e dimensão do assunto remete-se o leitor para as referências já habituais [Cla96, Mul98], e apresenta-se a seguir um exemplo comentado de uma das transformações mais simples, a *Transformação Identidade*, que copia para a saída o conteúdo do grove.

Exemplo 6-2. A Transformação identidade

```
; Transformação identidade
;    cria um novo grove igual ao primeiro
;

(=> (tudo) (transf-por-defeito))

; em que tudo é a Exp-Selecção

(define (tudo)
  (subgrove (current-root)))

; em que transf-por-defeito é a Exp-Transformação

(define (transf-por-defeito)
  (if (occurrence-mode (current-node))
      (transf-por-origem)
      (create-root #f (copy-current))))

(define (transf-por-origem)
  (create-sub
    (origin (current-node))
    (copiar-actual)
    (property: (occurrence-mode (current-node)))))

(define (copiar-actual)
  (subgrove-spec node: (current-node)))
```

Algumas legendas sobre as funções utilizadas:

subgrove

: devolve todos os descendentes de um nodo na forma de uma lista.

current-node

: devolve uma lista singular contendo o nodo que está a ser objecto da transformação.

subgrove-spec

o subGrove que está a ser criado é descrito utilizando um objecto do tipo *subgrove-spec*. O argumento *node:* especifica qual a raiz do subGrove que está a ser criado.

create-root

recebe como argumentos um identificador e um objecto do tipo *subgrove-spec* e especifica a criação da raiz do Grove final.

create-sub (origin...)

especifica a criação de um subGrove no Grove final com a origem especificada pelo argumento *origin*.

6.2.4.3. Geração de SGML

Esta fase é normalmente realizada por um gerador que faz uma travessia em profundidade (descendente, da esquerda para a direita) ao *Grove* enviando para o output o seu conteúdo em formato SGML. O documento SGML resultante pode, depois, ser passado a um formatador, ou usado para intercâmbio com outras aplicações.

6.2.5. Linguagem de Estilo

A linguagem de estilo permite descrever a aparência visual desejada para o documento final através da especificação do processo de formatação.

O processo de formatação vai:

- aplicar estilos de apresentação ao conteúdo do documento original e irá determinar a sua posição real na organização física das páginas.

- seleccionar e reordenar o conteúdo do documento final tendo em conta a sua posição no documento original.
- incluir material que não estava explicitamente presente no documento original - exemplo: geração de um índice.
- excluir, do documento final, material presente no documento original.

Uma especificação de estilo é composta por um conjunto de regras de construção. O objectivo destas regras é a construção de uma nova estrutura, a *Flow Object Tree (FOT)*. A FOT é uma árvore de objectos gráficos que funciona de representação intermédia para o processo de formatação. A especificação de estilo acaba por ser a especificação da transformação de uma árvore, o grove, noutra árvore, a FOT. Os objectos gráficos, que correspondem aos nodos da FOT encontram-se definidos no standard [Cla96]; no resto desta secção apresentam-se exemplos envolvendo a criação de alguns deles.

Actualmente o DSSSL suporta cinco tipos de regras de construção: *root*, *element*, *default*, *query* e *id*. Umas são mais específicas que as outras. Se as ordenássemos da mais para a menos específica obteríamos a seguinte ordem: *query, id, element, default, root*. A prioridade de uma regra serve para resolver conflitos, por exemplo: se um elemento do grove fôr seleccionado por uma regra do tipo *element* então, a regra *default* já não se aplicará áquele elemento.

6.2.5.1. Estrutura das Regras de Construção

Em DSSSL, as regras de construção seguem a estrutura genérica que se apresenta a seguir:

```
(nome-da-regra (exp-selecção)
               (exp-acção)
)
```

A *exp-selecção* serve para seleccionar os nodos do *Grove* aos quais será aplicada a *exp-acção*. Cada exp-acção está normalmente associada à construção dum nodo da FOT. Cada nodo da FOT pertence a uma classe de objectos gráficos. Cada classe tem a ela associados uma série de atributos que vão determinar a aparência visual dos respectivos objectos.

O DSSSL possui um conjunto de classes base e dá ao utilizador a possibilidade de definir outras.

Apresentam-se a seguir vários exemplos de regras de construção.

Exemplo 6-3. Regra associada ao documento

```
(root
  (make display-group
    (process-children)
  )
)
```

Aqui, *root* corresponde ao nome da regra, a expressão de selecção encontra-se omissa, e a acção corresponde à invocação da função *make*.

A acção *process-children* indica que se pretende aplicar o resto da especificação aos descendentes deste nodo. Sem esta indicação o documento seria truncado a partir deste nodo.

Exemplo 6-4. Regra associada a todos os elementos no documento original

```
(default
  (process-children)
)
```

Neste exemplo, *default* corresponde ao nome da regra, a expressão de selecção é omissa, e a acção corresponde à invocação da função *process-children*.

As duas regras apresentadas em cima fogem ao caso genérico ao não terem a expressão de selecção. Isto deve-se ao facto de essa expressão estar subjacente ao tipo de regra. No primeiro caso, regra de construção *root*, a regra, se estiver presente, é sempre seguida, mas apenas uma vez, quando o elemento a processar é a raiz do grove, que representa todo o documento. No segundo caso, regra de

construção *default*, a regra é seguida para todos os nodos elemento, excepto quando estes são seleccionados por outra regra (lembrar a especificidade).

Exemplo 6-5. Regra associada a uma classe de elementos

```
(element (seccao titulo)
  (make paragraph
    font-family-name: "Helvetica"
    font-weight: 'bold
    (process-children)
  )
)
```

A expressão de selecção *(seccao titulo)* indica que esta regra será usada para processar nodos *titulo* que são filhos de nodos *seccao*. Esta regra ficará associada a todos os elementos do documento que pertençam a esta classe. Outros nodos *titulo*, que não dentro de secção, não serão associados a esta regra.

Exemplo 6-6. Regra associada a um elemento específico identificado por um atributo do tipo ID

```
(id ("sec2")
  (make paragraph
    font-family-name: "Helvetica"
    font-weight: 'bold
    (process-children)
  )
)
```

Ao contrário da regra anterior, esta fica associada apenas a uma instância dum elemento, áquele que tiver um atributo do tipo ID com valor igual ao indicado na expressão de selecção - *"sec2"*.

Exemplo 6-7. Regra associada a uma ’query’

```
(query q-class ’pi
  (make paragraph
    literal "Instrução de Processamento:"
    (node-property ’system-data (current-node))
  )
)
```

Esta regra permite seleccionar todos os elementos abrangidos pela expressão de selecção. Neste caso, a expressão de selecção *’pi* indica que se pretende seleccionar todas as *"Processing Instructions"*. Devido à grande complexidade que levanta na sua implementação, não foi implementada nos processadores de DSSSL actualmente disponíveis.

6.2.6. O Processo de Formatação

Tal como se disse atrás para o caso da transformação, antes do início do processamento o documento tem que ser analisado e uma estrutura arbórea tem que ser criada com o seu conteúdo - o Grove: se o processo de formatação se seguir a uma transformação o grove encontra-se já disponível; senão, terá de se começar pelo parsing do documento fonte. Como já se mostrou, cada nodo desta estrutura corresponde a um elemento do documento. O processo de formatação especifica-se associando a cada nodo do grove um conjunto de regras de construção que irão criar uma nova estrutura - a *Flow Object Tree (FOT)*. Nesta nova estrutura, cada um dos nodos corresponde a um objecto com características gráficas. Para finalizar o processo, esta estrutura é passada a um formatador que apenas vai organizar espacialmente, no suporte físico de output seleccionado, os objectos gráficos.

Assim, e resumindo, o processo de formatação, que se vê esquematizado na Figura 6-3, compreende três fases:

Construção do grove

O documento SGML é submetido a um parser que o vai validar e produzir um Grove (como no processo de transformação).

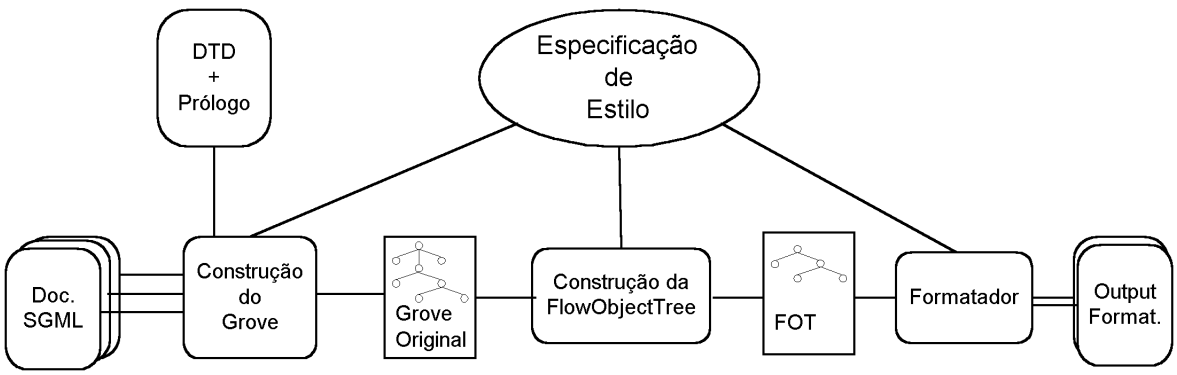
Construção da Flow Object Tree

O grove é submetido a um processo que lhe vai aplicar as regras de construção especificadas na especificação de estilo e assim gerar a *Flow Object Tree*.

Formatação

O formatador parte da *Flow Object Tree*, aplica as suas regras geométricas, faz a composição dos vários objectos gráficos pertencentes à *Flow Object Tree* e produz o documento final.

Figura 6-3. DSSSL - processo de formatação



6.2.7. Algumas especificações exemplo

Apresentam-se agora algumas especificações DSSSL que, se fossem utilizadas para processar o livro de receitas do Exemplo 6-1, serviriam para obter os efeitos indicados em cada caso.

Comecemos por uma especificação de estilo básica que simplesmente converterá o texto do documento SGML original para RTF, MIF, ou TeX (conforme a opção passada ao formatador), sem nenhum requisito de formatação.

Exemplo 6-8. Especificação de Estilo básica

```
(root
  (make simple-page-sequence
    (process-children))
)
```

- *make* é uma função que constrói um nodo da FOT; neste caso um nodo da classe *simple-page-sequence*.
- um nodo do tipo *simple-page-sequence* vai fazer com que seja produzida uma sequência de áreas de página.
- a função *process-children* dá a indicação de que o processamento deverá continuar para os descendentes do nodo que se está a processar.

Vamos agora complicar um pouco o exemplo anterior, adicionando à especificação as indicações necessárias para formatar as margens do texto.

Exemplo 6-9. Adicionando margens

```
(root
  (make simple-page-sequence
    left-margin:    3cm
    right-margin:   2cm
    top-margin:     3cm
    bottom-margin:  3cm
    (process-children))
)
```

O objecto *simple-page-sequence* tem uma série de atributos que permitem descrever a formatação do seu conteúdo. Neste caso utilizamos os atributos relativos às dimensões das páginas.

Complicando mais um pouco, vamos agora formatar de maneira diferente o título do livro de receitas e o título individual de cada receita.

Exemplo 6-10. Formatando os títulos

```
(element (RECEITAS TITULO)
  (make paragraph
    quadding:        'center
    font-size:       18pt
    keep-with-next?: #f
    (process-children)
  )
)

(element (RECEITA TITULO)
  (make paragraph
    quadding:        'left
    font-size:       16pt
    keep-with-next?: #f
    (process-children)
  )
)
```

- As expressões de selecção associam respectivamente uns atributos a um e ao outro tipo de titulo.
- A classe paragraph é geralmente utilizada para elementos do tipo bloco, que se individualizam dos elementos que o precedem e que lhe sucedem.

A especificação anterior pode ser reescrita recorrendo à utilização de variáveis para os valores dos atributos, o que facilitará a sua manutenção.

Exemplo 6-11. Adicionando variáveis para facilitar a manutenção

```
;Constantes
```

```
(define *tituloPrinc* 18pt)
(define *tituloReceita* (- *tituloPrinc* 2))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(element (RECEITAS TITULO)
  (make paragraph
    quadding:        'center
    font-size:       *tituloPrinc*
    keep-with-next?: #f
    (process-children)
  )
)

(element (RECEITA TITULO)
  (make paragraph
    quadding:        'left
    font-size:       *tituloReceita*
    keep-with-next?: #f
    (process-children)
  )
)
```

Desta maneira, a modificação do tamanho da fonte só necessita de ser feita num local, o que facilita enormemente a manutenção.

Aqui estão a ser utilizados alguns dos muitos atributos da classe *paragraph*. Para uma informação mais detalhada sobre as classes e respectivos atributos aconselha-se a leitura do standard.

Por fim, mostra-se um exemplo que exibe o tipo de especificação utilizada para a criação de índices e outros tipos de listas de conteúdos. Neste caso, iremos colocar no início do livro de receitas uma lista de todos os ingredientes necessários para a execução de todas as receitas - isto implica uma passagem adicional sobre o documento para a construção desta lista.

Exemplo 6-12. Utilizando mais de uma passagem sobre o documento

```
(element RECEITAS
  (make sequence
    (literal "Ingredientes necessários:")
    (make paragraph
      (with-mode *ingredientes*
        (process-matching-children "INGREDIENTE")))
    (process-children)
  )
)

(mode *ingredientes*
  (element INGREDIENTE
    (if (= (child-number) 1)
      (make sequence
        font-posture: 'italic
        (process-children))
      (make sequence
        font-posture: 'italic
        (literal " & ")
        (process-children))))
  (default
    (empty-sosofo)))
```

Cada passagem sobre o documento tem a designação de *mode*. Se nada fôr declarado o processamento é realizado em *normal-mode*; para processamentos adicionais outros *mode* terão de ser criados - neste caso criou-se o *mode ingredientes*.

Nesta secção, deu-se uma ideia da potencialidade do DSSSL e da metodologia de especificação que lhe é subjacente. Muita coisa ficou fora desta abordagem, no entanto, as características aqui apresentadas permitem concretizar processos de formatação com alguma complexidade.

A Semântica Estática é um assunto completamente novo. Até à data, não houve nenhum estudo que apontasse uma solução para a sua definição e implementação. Neste momento, existem umas propostas em estudo [DCD, WIDL] a nível do *World*

Wide Web Consortium (W3C) - entidade que superintende a criação de novos standards relacionados com a publicação electrónica e a Internet, mas numa forma ainda muito incipiente.

Como a discussão desta temática reflecte a contribuição mais significativa desta tese, é apresentada num capítulo à parte, o próximo, Capítulo 7.

Notas

1. No contexto de uma disciplina de compilação leccionada às licenciaturas em informática, na Universidade do Minho.

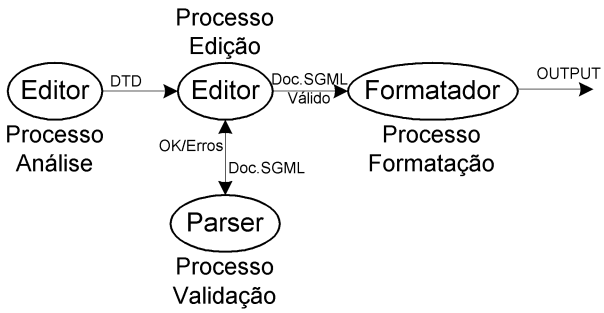
Capítulo 7. Validação Semântica em Documentos SGML

7.1. Semântica Estática

A introdução do SGML trouxe uma nova abordagem ao processamento documental: validação estrutural automática, validação em tempo real durante a edição (o autor está sempre informado sobre a correcção ou incorrecção estrutural do documento que está a produzir), independência face a plataformas de software e hardware.

Na figura seguinte mostra-se este modelo de trabalho, que não é mais do que um subconjunto do ciclo de vida apresentado na Capítulo 5.

Figura 7-1. Edição baseada em SGML



7.1.1. Qual é o problema?

O modelo apresentado é suficiente para uma grande parte das aplicações do SGML. No entanto, quando se trabalha um certo tipo de informação há uma grave lacuna: *validação semântica*. Por exemplo, suponhamos que alguém está a produzir um CD-ROM com conteúdo histórico; um erro na data de nascimento dum rei ou uma associação cronológica com o reinado mal feita pode fazer colapsar todo o projecto.

Relativamente à consistência e correcção da informação, o SGML assegura uma validação estrutural completa e alguma validação semântica (através da associação

de atributos aos elementos); é possível garantir que um determinado atributo tem um valor dentro dum conjunto de possibilidades. Mas, quando trabalhamos com documentos com conteúdos mais ricos precisamos de colocar restrições mais fortes sobre esses conteúdos; esta necessidade está para além das capacidades do SGML.

Esta lacuna tornou-se evidente quando se avançou na implementação de projectos reais (descritos mais à frente em Apêndice B). Nesses projectos, trabalhámos com o Arquivo Distrital de Braga e com o Departamento de História da Universidade do Minho. Pretendia-se disponibilizar grandes volumes de informação na Internet e trabalhá-la de modo a ser possível gerar também outras formas: CD-ROM, papel, ... Para tal montou-se uma linha editorial baseada em SGML. Um dos problemas críticos era o tempo e aqui começou a sentir-se a falta de uma validação semântica automática que, como era inexistente, levou a que se implementasse uma série de ciclos de revisão, o que atrasou (e ainda atrasa) os projectos na sua fase final de divulgação da informação.

Nas secções seguintes, não iremos apresentar uma solução completa para o problema da validação semântica, a solução desenvolvida tratará um subconjunto de problemas de validação semântica: restringir o conteúdo textual (#PCDATA) de um determinado tipo de elemento, e verificar relações entre subelementos e atributos.

Para exemplicar estas ideias apresentam-se a seguir dois casos de estudo. Estes dois casos emergem dos projectos acima referidos, onde estamos a colectar informação de várias fontes, que depois de processada é distribuída via Internet. Os problemas apontados são simples e seriam facilmente resolvidos se houvesse alguma maneira de expressar restrições no modelo.

7.1.1.1. Registos Paroquiais

Neste caso, a fonte de informação são os registos duma paróquia. Numa comunidade católica, o pároco mantém um registo dos eventos realizados na paróquia: casamentos, baptismos, funerais, ... Temos portanto, vários tipos de documentos: certidões de casamento, de baptismos, de óbito, e ainda outros que não se encaixam numa classificação deste tipo.

O que se pretende aqui, é reunir aquela informação documental e construir uma base de dados de indivíduos com as respectivas relações familiares. Esta base de dados poderá depois fornecer informação para a construção de modelos estatísticos que registarão a evolução de hábitos e costumes permitindo que cada um de nós

possa aprender um pouco sobre as suas raízes.

Imagine-se agora, que temos uma equipe a introduzir aqueles documentos no computador em formato SGML (os originais estão bastante deteriorados e são demasiado irregulares na estrutura para permitirem uma digitalização automática). As pessoas, por vezes, cometem erros nas transcrições. O parser de SGML detecta os erros de sintaxe/estruturais. Os outros, erros semânticos, passam indetectados e poderão levar a vários problemas que influenciarão os modelos estatísticos que se pretendem criar. Por exemplo:

- idades ao casamento negativas - provavelmente foi introduzida uma data errada num dos certificados.
- morte antes do baptismo - o mesmo problema retratado acima.
- casamentos entre pessoas com uma diferença de idades superior a 100 - um dos elementos do casal deve ter uma data a ele associada que foi mal introduzida.

Todos estes problemas podiam ser resolvidos com restrições simples sobre o domínio de valores: "data de nascimento deve ser sempre maior que data de morte"; relativamente a casamentos, "a diferença das datas de nascimento dos conjugues deverá ser sempre menor que um determinado valor", etc.

7.1.1.2. Arqueologia

Outra fonte de informação com quem trabalhamos é a Unidade de Arqueologia da Universidade do Minho.

Esta equipe de arqueólogos tem produzido uma série de documentos SGML que retratam arqueosítios e artefactos. Na estrutura destes documentos há 2 elementos que têm de estar presentes obrigatoriamente e que dizem respeito à localização geográfica da entidade descrita: *latitude e longitude*. À medida que cada documento é produzido queremos ligá-lo a um ponto dum mapa, construindo desta forma um Sistema de Informação Geográfica (SIG).

Uma coisa que queremos e devemos garantir é que cada par de coordenadas cai dentro do mapa, dentro dum certo intervalo. Isto, não pode ser feito usando apenas SGML, mas uma linguagem simples onde se pudessem expressar restrições sobre um dado domínio serviria.

7.1.2. Restrições e condições de contexto

Temos um problema para resolver. Queremos impôr condições sobre o conteúdo de alguns elementos e queremos que o sistema tenha capacidade para verificar essas condições. Para isso, precisamos de adicionar restrições aos documentos SGML, o que traz algumas implicações e levanta outros problemas: normalização de conteúdos, associação de tipos aos conteúdos, que linguagem utilizar para definir as restrições, e como processá-las.

Como já foi referido, não se pode utilizar a sintaxe do SGML para expressar restrições ou invariantes sobre o conteúdo dos elementos que compõem o documento. O SGML foi criado como uma metodologia para especificação de estrutura, e na área da validação estrutural o grande número de ferramentas disponíveis faz dele uma forte e poderosa metodologia de especificação.

Se o que pretendemos para salvaguardar a semântica dos documentos é restringir conteúdos, precisamos de facto de adicionar essa capacidade ao SGML. Para isso, temos várias hipóteses: pode-se simplesmente adicionar sintaxe extra ao SGML; ou desenhar uma nova linguagem que podia ser embebida no SGML, ou coexistir fora dele. A adição de sintaxe extra implicaria a modificação de todas as ferramentas existentes para que estas reagissem à nova extensão ou simplesmente a ignorassem. O custo desta solução seria enorme; por isso seguiremos o outro caminho: a definição duma linguagem que permita a especificação de restrições sobre elementos em documentos SGML.

Até ao momento, as restrições que sentimos necessidade de impôr são bastante simples. Na maioria dos casos, apenas se pretende restringir o valor de elementos atómicos a um domínio, verificar uma relação entre elementos, ou verificar se um determinado conteúdo está coerente com a informação guardada numa base de dados de suporte. Uma das razões para isto acontecer é a validação estrutural já realizada pelo SGML ser tão forte que apenas deixa o conteúdo dos elementos de fora.

Pode parecer, portanto, que uma linguagem simples resolve o problema. No entanto, podemos desde já distinguir duas fases importantes neste processo:

- *a definição* - a parte sintáctica do modelo de restrições, as declarações que expressam as restrições.
- *o processamento* - a parte semântica do modelo de restrições, a sua interpretação.

Estas duas fases têm objectivos diferentes e correspondem também a diferentes níveis de dificuldade na sua implementação. A primeira envolve a criação de uma linguagem, ou a adopção duma existente. Para a fase de processamento, precisamos de implementar um motor que seja capaz de avaliar e reagir às declarações escritas na linguagem de restrições. Para implementar este motor vamos necessitar de ter tipos associados à nossa informação com toda a complexidade que lhes é inerente.

Neste momento, podíamos ser tentados a questionar esta complexidade: "Não poderemos sobreviver sem tipos?". Consideremos o seguinte exemplo extraído do segundo caso de estudo (Secção 7.1.1.2):

Exemplo 7-1. SGML com restrições

```
SGML Document
  <latitude>41.32</latitude>

Restrição
  latitude > 39 and latitude < 43
```

Neste exemplo, queremos garantir que o valor introduzido no elemento latitude de um arqueosítio está compreendido dentro de certos valores. Estamos a verificar se um determinado valor está dentro dum domínio. Estamos a comparar o conteúdo do elemento latitude com valores numéricos que têm um tipo inerente (inteiro ou real). Assim, o motor que vai executar esta comparação irá ter de inferir um tipo para o conteúdo do elemento latitude que está a ser comparado.

Exemplos como este são muito simples. O conteúdo numérico de um elemento tem uma forma mais ou menos normalizada (pode ser expressso por uma expressão regular). Mas há outros bem mais complexos, como a data: há mais de cem maneiras diferentes de escrever uma data (e provavelmente qualquer um de nós pode facilmente inventar mais uma).

Noutro caso de estudo, um trabalho realizado para o Arquivo Distrital de Braga [Fig98], estamos a trabalhar com documentos do século XVII e XVIII; nesses documentos, o nome "Afonso Henriques", o primeiro rei português, aparece escrito de duas maneiras diferentes: "Afonso"e "Affonso". E ainda há outros personagens que são referenciados de mais de duas maneiras distintas. Para podermos impôr

restrições ao nome dos reis de Portugal, temos de saber o tipo desses elementos e de usar um valor independente da ortografia.

Vamos designar este problema por *problema de normalização*. Para além desta situação de imposição de restrições, a normalização torna-se crítica quando se pretende construir índices sobre os documentos, para alimentar motores de busca ou simplesmente para implementar mecanismos de acesso à informação. O computador precisa de entender que textos diferentes são referências para o mesmo objecto.

Por agora, propõem-se duas soluções para resolver os problemas da normalização e da inferência de tipos:

- desenvolver e implementar uma linha de ferramentas, com alguma complexidade, que resolverão os problemas, primeiro a normalização, depois a inferência de tipos.
- introduzir algumas modificações no DTD que resolverão o problema da normalização e reduzirão a inferência de tipos a um problema mais simples.

Obviamente, é a segunda que deve ser seguida. Não só porque é mais simples, mas também porque o problema da normalização tem atraído a atenção de investigadores, principalmente na área da História, o que fez com que várias aproximações a uma boa solução tivessem surgido. Uma das melhores e a mais utilizada no momento é a proposta do "Text Encoding Initiative (TEI)"[SB94]. A solução é bastante simples, basta adicionar um atributo de nome *"value"* (no nosso caso designaremos esse atributo por *"valor"*) aos elementos cujo conteúdo levantará problemas de normalização. A ideia é a do conteúdo do elemento poder tomar qualquer forma, desde que o atributo *"valor"* seja preenchido com o valor normalizado.

Exemplo 7-2. Normalização via adição dum atributo

```
... perdeu a batalha no <data valor="1853.10.05»
quinto dia do mês de Outubro do ano 1853 </data> ...
```

Neste exemplo, convencionou-se que o formato normalizado para as datas seria o ANSI (ano.mês.dia).

Esta solução, pressupõe uma análise prévia do conteúdo documental, que pode ser feita antes ou depois da edição do documento em SGML. Durante essa análise deverão ser levantados todos os problemas de normalização para que depois ou durante a edição do documento SGML se adicione, aos elementos críticos, o atributo *"valor"* com o valor convencionado.

Em trabalhos publicados durante a realização desta tese [RRAH99], propôs-se uma solução semelhante para simplificar a inferência de tipos. A todos os elementos que irão ter a eles associada uma restrição é adicionado um atributo de nome "type"("tipo"na implementação portuguesa) cujo valor é a designação do tipo de dados que se quer associar aos elementos em causa.

Se aplicarmos estas duas receitas aos exemplos discutidos durante esta secção o resultado seria o seguinte:

[latitude]

```
            <latitude tipo="real» 41.32 </latitude> ...
```

[datas]

```
            ... aconteceu no <data tipo="date" va-
lor="1853.10.05»
      quinto dia do mês de Outu-
bro do ano 1853 </data> ...
```

[nomes]

```
            ... naquele ano <nome valor="Afonso» Affon-
so </nome>
      proclamou vários decretos ...
```

Assume-se que, quando um atributo *"valor"* não se encontra instanciado, o conteúdo do elemento respectivo já se encontra na forma normalizada.

Relativamente ao atributo *"tipo"*, a sua colocação levanta alguns problemas. Normalmente, o analista que desenvolve o DTD e o utilizador que vai usar esse DTD para editar os seus documentos são pessoas diferentes, com níveis de conhecimento técnico diferentes. E, este atributo faz com que o utilizador tenha que saber o que são tipos de dados, quais os tipos de elementos que irão ter restrições

sobre eles (só esses necessitam de ser "tipados"), e que instancie correctamente este atributo para estes elementos. O ideal seria que apenas o analista se preocupasse com este aspecto e tudo isto fosse transparente para o utilizador. O SGML tem uma característica que nos permite implementar esta separação: os atributos do tipo *"#FIXED"*.

Sempre que tivermos um elemento com um determinado atributo de valor constante, e não quisermos que o utilizador altere esse valor, esse atributo deve ser declarado como *#FIXED* no DTD onde o respectivo valor deverá também ser declarado.

Voltando ao exemplo das latitudes:

Exemplo 7-3. Atributos "#FIXED"

```
DTD
  ...
  <ELEMENT latitude  - - (#PCDATA)>
  <ATTLIST latitude  tipo CDATA #FIXED "real»
  ...
```

- *tipo* é o nome do atributo.
- *CDATA* é o tipo do valor do atributo, neste caso, indica que será uma string.
- *#FIXED* é o valor por omissão do atributo; indica ao sistema que no caso do utilizador não tiver sido instanciado o sistema deverá assumir o valor fornecido na declaração, por outro lado, se o utilizador instanciar o atributo o sistema deverá garantir que este é idêntico áquele que foi fornecido na declaração.
- *"real"* é o valor do atributo.

Ao trabalhar com um documento que esteja a ser criado com o DTD exemplificado acima, o sistema, em todas as operações que envolvam o processamento SGML (por exemplo na exportação do documento para o sistema), acrescentará um atributo *"tipo"*a todos os elementos *"latitude"* com valor *"real"*. Qualquer tentativa do utilizador para instanciar aquele atributo com outro valor será ignorada pelo

sistema. Desta maneira, a associação de tipos aos elementos tornou-se transparente para o utilizador, até mesmo inacessível.

A adição destes dois atributos a um ou a vários elementos levanta alguns problemas nas futuras transformações que se pretendam implementar sobre o documento. No caso do atributo valor, há situações em que se pretende utilizar o valor normalizado (por exemplo numa procura, ou na construção dum índice), e outras, em que se pretende utilizar o valor original (por exemplo numa publicação fiel ao documento original).

Uma questão relevante pode agora ser levantada: Será que necessitamos de associar um tipo a todos os elementos? Ou apenas a elementos atómicos (aqueles cujo conteúdo é apenas #PCDATA)?

A questão levantada aparenta ser simples com uma resposta binária de sim ou não. No entanto, qualquer uma das respostas traz consigo um certo peso.

Uma resposta afirmativa implicaria que para além de tipos atómicos de dados, o nosso sistema, suportasse tipos estruturados que seriam associados aos nodos intermédios da árvore documental. Desta maneira, teríamos um mapeamento completo entre a estrutura do documento e o nosso modelo abstracto de tipos. As respectivas consequências desta abordagem, vantagens e desvantagens, seriam:

- o sistema de tipos fica mais complexo.
- os tipos estruturados seriam mais facilmente inferidos do DTD do que do conteúdo dos elementos; isto implicaria a criação de uma ferramenta de conversão.
- um mapeamento completo entre a estrutura do documento e um modelo abstracto de tipos de dados permitiria o processamento do documento no sistema de suporte do modelo abstracto; o que implicaria a criação e a utilização de ferramentas poderosas (estas ferramentas resultariam da combinação dos operadores algébricos do sistema de suporte do modelo abstracto).

Por outro lado, se decidíssemos associar um tipo de dados apenas a alguns elementos atómicos (algumas folhas da árvore documental), poderíamos prever as seguintes consequências:

- a linguagem de restrições seria muito simples; reduzida a operações lógicas sobre tipos atómicos de dados e algumas funções de interrogação.

- o motor de processamento resultaria simples e por isso fácil de implementar.
- o modelo abstracto estaria incompleto; ficaríamos com um conjunto de pequenos bocados do modelo completo; isto tornaria inviável qualquer processamento do documento no sistema de suporte do modelo abstracto.

Em defesa deste último ponto de vista podemos ainda apontar o seguinte: o SGML é puramente declarativo ("declarative markup"); serve para definir estruturas; estas estruturas não são mais nem menos do que os tipos estruturados que temos vindo a discutir. Assim, podemos afirmar que os tipos estruturados são intrínsecos aos documentos; a respectiva validação é feita quando o documento é analisado e portanto não será necessário complicar a linguagem de restrições para realizar esta tarefa. Para clarificar este ponto, veja-se o próximo exemplo: vamos descrever em SGML uma árvore binária que neste caso modela uma árvore genealógica.

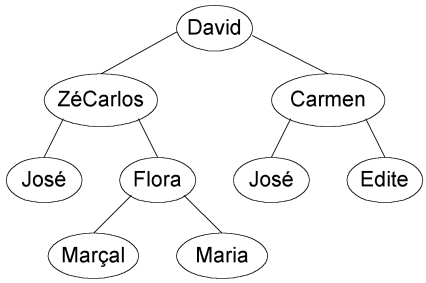
Exemplo 7-4. Uma árvore binária

O respectivo DTD (a declaração da estrutura de dados) define-se da seguinte maneira:

```
<!DOCTYPE arv-bin [
<!ELEMENT arv-bin - - (nodo)>
<!ELEMENT nodo    - - (valor,filhos?)>
<!ELEMENT valor   - - (#PCDATA)>
<!ELEMENT filhos  - - (nodo?,nodo?)>
]>
```

O leitor atento reparará que se introduz um nível de redundância na especificação ao declarar-se o elemento FILHOS como opcional e, depois, os seus descendentes NODO, de novo, como opcionais. Isso foi feito conscientemente devido às implicações que existiriam na escrita de instâncias se FILHO, fosse obrigatário (para nodos sem filhos o autor seria sempre obrigado a colocar a anotação FILHO apesar de depois não lhe introduzir nenhum conteúdo).

A figura seguinte apresenta uma árvore genealógica que a seguir é instanciada em SGML de acordo com o DTD apresentado.




```
<arv-bin>
  <nodo>
    <valor>David</valor>
    <filhos>
      <nodo>
        <valor>ZéCarlos</valor>
        <filhos>
          <nodo>
            <valor>José</valor>
          </nodo>
          <nodo>
            <valor>Flora</valor>
            <filhos>
              <nodo>
                <valor>Marçal</valor>
              </nodo>
              <nodo>
                <valor>Maria</valor>
              </nodo>
            </filhos>
          </nodo>
        </filhos>
      </nodo>
      <nodo>
        <valor>Carmen</valor>
        <filhos>
          <nodo>
            <valor>José</valor>
          </nodo>
          <nodo>
            <valor>Edite</valor>
          </nodo>
```

```
      </filhos>
    </nodo>
  </filhos>
</nodo>
```

Deste exemplo, podemos concluir o seguinte: podemos definir tipos estruturados de dados duma forma puramente declarativa; a semântica está nos tipos atómicos; se ignorarmos os tipos atómicos ou, deixarmos ao sistema o trabalho de inferir o seu tipo, podemos afirmar que as estruturas definidas em SGML correspondem a tipos estruturados de dados.

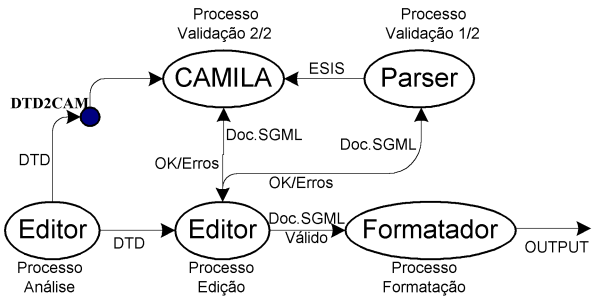
Num primeiro trabalho publicado no âmbito desta tese [RAH95], apresentou-se uma abordagem à primeira das direcções apontadas (associação de tipos a todos os elementos). Num trabalho posterior [RAH96], aquela abordagem foi refinada e um estudo comparativo com uma outra foi apresentado. Como resultado desse estudo, desenvolveram-se duas soluções para o problema do processamento da semântica estática. Cada solução seguiu uma abordagem formal diferente. São estas duas abordagens que se apresentam nos dois capítulos seguintes e que são respectivamente, uma abordagem via modelos abstractos, Capítulo 8, e uma abordagem via gramáticas de atributos, Capítulo 9.

Capítulo 8. Validação Semântica com Modelos Abstractos

Neste capítulo vamos apresentar uma solução para o processamento de restrições semânticas em que esse processamento residirá num ambiente externo ao da aplicação SGML. A ideia é criar uma solução compatível com as aplicações existentes, fazer com que o processamento das restrições se possa acoplar ao que já existe sem alterações das metodologias e ferramentas de trabalho.

Como se pode ver na figura seguinte (Figura 8-1), aquilo que se propõe aqui fazer é uma pequena adição ao modelo de processamento existente: um novo componente que não provocará nenhuma alteração dos outros componentes.

Figura 8-1. Novo Modelo para Processamento de Documentos SGML



Neste capítulo, iremos ver como foi implementado este novo componente e, o impacto e respectivas consequências no ciclo de vida da produção de documentos.

8.1. Modelos Abstractos: porquê?

Para ilustrar estas ideias iremos utilizar um caso de estudo, bastante importante para nós (é bom para demonstrar vários aspectos do problema): o tipo de documentos subjacente ao conceito "*literate programming*"[Knu92]. Este tipo de documentos foi criado para permitir a mistura, dentro do mesmo ficheiro, do código de um programa e respectivo manual/relatório. Para isso estão disponíveis anotações que

permitem a identificação e o estabelecimento de relações entre aqueles dois tipos de componentes, de modo a que mais tarde seja possível extrair as partes relevantes: o programa para compilar ou o relatório para imprimir. Com efeito, nos exemplos que aparecem à frente, iremos centrar a nossa atenção no procedimento de extracção e composição de programas a partir dum documento *literate programming*. Este procedimento servirá como termo de comparação das várias abordagens.

Assim, um documento em *literate programming* é um texto de conteúdo misto, composto por elementos especiais, código e definições. Uma definição é uma associação de um identificador a um pedaço de código ou texto; um programa pode incluir referências a estes identificadores.

Apresenta-se a seguir um exemplo dum DTD e respectiva instância documental para este tipo de documentos.

Exemplo 8-1. Literate Programming

O DTD que se segue define formalmente a estrutura dum documento do tipo *literate programming*.

```
          <!DOCTYPE litprog [
          <!ELEMENT litprog - -
((#PCDATA|prog|def|id|sec|tit)*)>
          <!ELEMENT prog    - - ((#PCDATA|id)*)>
          <!ELEMENT def     - - (prog)>
          <!ATTLIST def ident ID #REQUIRED>
          <!ELEMENT sec     - - (#PCDATA)>
          <!ELEMENT tit     - - (#PCDATA)>
          <!ELEMENT id      - o EMPTY>
          <!ATTLIST id refid IDREF #REQUIRED>
          ]>
```

O texto seguinte é um exemplo concreto dum documento escrito de acordo com o DTD acima.

```
          <litprog>
            <tit>Exemplo de Literate Programming</tit>
            <sec>Stack - FAQ</sec>
            <def ident="main»
              <prog>
```

```
                main(){
                int S[20]; sp=0;
                <id refid="push»
                <id refid="pop»
                }
              </prog>
            </def>

            <sec>Push</sec>
            Esta função pode-
se usar para colocar elementos na stack.
            <def ident="push»
              <prog>void push(int x)
                      {S[sp++]=x;}
              </prog>
            </def>
          </litprog>
```

O texto em si é um manual dum determinado programa. Quando se produz um documento deste tipo há dois processamentos esperados: a produção do manual propriamente dito e a extracção do programa para se compilar. O extractor poderá retirar o programa ou parte dele, o utilizador indicará o que pretende passando o identificador da raíz da subárvore de pedaços de código que pretende extrair.

Nesta aproximação iremos converter o documento num modelo abstracto de dados e utilizaremos o sistema de suporte desse modelo (neste caso o CAMILA) para especificar o processamento do documento.

Um DTD, do ponto de vista algébrico pode ser visto como um modelo, como já foi discutido em [RAH95]. Por exemplo, no nosso caso de estudo teríamos o seguinte modelo CAMILA:

```
model
type
  litprog = X-seq
  X       = (TEXTO | prog | def
            | id | sec | tit)

  prog    = Y-seq
```

```
  Y       = (TEXTO | id)

  def     = i:ID * p:prog
  sec     = TEXTO
  id      = ID
  tit     = TEXTO
endtype
```

A construção do modelo CAMILA foi feita sistematicamente seguindo a tabela de equivalência abaixo apresentada.

Tabela 8-1. Esquema de Tradução SGML ↔ CAMILA

SGML	CAMILA
x,y	produto cartesiano
x & y	produto cartesiano
x | y	união disjunta
x*	X-seq
x+	X-seq
x?	[X]

Desta tabela é fácil concluir que não há uma equivalência directa entre os elementos dos dois universos. Na maior parte dos casos, para não se perder semântica, teria de ser associado um invariante ao tipo de dados CAMILA. Por exemplo, no quinto caso (x+), no correspondente tipo CAMILA nada é dito sobre a cardinalidade da sequência; um invariante garantindo que a sequência não é vazia teria de ser associado ao tipo CAMILA correspondente.

Nesta abordagem, o processamento é especificado como uma função ou funções sobre o modelo abstracto. Por exemplo, o nosso extractor de programas *getprogram()*, será então expresso como uma função que recebe um documento do tipo *literate programming*, colecta todos os bocados de código nele espalhados, organiza-os de forma correcta, produzindo assim um programa, neste caso em C, do tipo *cprog* definido abaixo.

```
type
  cprog = TEXTO-seq
endtype
```

Então, o referido extractor poderia ser assim especificado:

```
func getprogram( d :litprog ) :cprog
returns explode('main,mkindex(t));
```

A função *getprogram()* realiza uma substituição recursiva (*explode*) dos identificadores (*id*) pelos respectivos programas, começando pelo identificador *main*. O exercício fica completo com a definição das duas funções que faltam: *explode()* e *mkindex()*.

```
func mkindex( d :litprog ) :id->prog
return [i(x) -> p(x) | x <- t : is-def(x) ];

/* a função mkindex recebe um documento e cons-
troi uma
   função finita de identificador para boca-
do de código
   -- é a função colectora */

func explode( i :id, d :id->prog ) :cprog
pre i in dom(d)
returns( CONC( < if( is-id(x) -> explode(x,d),
                 else -> <x> ) | x <- d[i]> );

/* a função explode recebe a função finita construí-
da na
   função anterior, um identificador indican-
do qual a raíz
   da derivação, e constrói o programa - é a função
   compositora */
```

Esta abordagem é boa para prototipar, quer estruturas, quer processamento. No caso específico dos documentos SGML, apresentamos uma possível via de translacção do seu processamento para o nosso sistema algébrico. Há no entanto alguns pontos a refinar: o esquema de tradução tem de ser enriquecido, os modelos abstractos nos quais mapeamos os construtores do SGML são, regra geral, menos específicos, têm menor poder expressivo; assim, terão de ser restringidos via adição de um invariante de tipo para se conseguir manter a mesma semântica. A maior parte da especificidade do SGML resulta deste ter sido pensado para anotar texto. O texto tem uma ordem linear inerente. Esta ordem foi transportada para o SGML sob a

forma dos operadores "&"e ",". Se tivermos um DTD onde aqueles operadores existam, a sua conversão num modelo CAMILA necessitará da adição de invariantes que garantam a especificidade da ordem.

Poderíamos ser tentados neste ponto a pensar se não valeria a pena criar um protótipo em CAMILA que suportasse o ciclo de vida do SGML. A ideia seria prescindirmos do parser e usar os constructores do CAMILA para especificar uma estrutura que depois seria preenchida com instâncias documentais às quais seriam aplicados processamentos sob a forma de execução de funções sobre o modelo algébrico. Isto seria possível se a equivalência semântica entre as duas notações de especificação fosse real. Como já foi discutido a notação CAMILA é mais lata, menos específica que o SGML e a criação deste sistema só seria possível via adição de vários invariantes e condições de contexto que no seu todo seriam semanticamente equivalentes a um parser SGML.

8.2. Linguagem de Restrições

A questão pertinente do momento é: Como iremos especificar as restrições? Em que linguagem?

Como já foi discutido no capítulo passado, temos duas opções: desenvolver uma nova linguagem, ou, utilizar uma existente. Uma vez que o que se está a desenvolver é um protótipo e a escolha de uma linguagem existente poupar-nos-ia algum trabalho, foi esta a opção escolhida.

Para se poder manipular elementos SGML e especificar restrições sobre eles a escolha natural seria uma linguagem de especificação baseada em modelos. Como já foi exemplificado na secção anterior (Secção 8.1) e em [RAH95], cada definição dum elemento no DTD tem um modelo implícito, e cada instância desse elemento pode ser convertida numa expressão algébrica desse modelo. No protótipo descrito mais à frente Figura 8-2, estamos a utilizar uma ferramenta de conversão automática, *dtd2cam*, para traduzir o DTD para um modelo na linguagem de especificação de SETs, CAMILA [ABNO97, BA95]. Depois desta conversão, o analista pode escrever as restrições em CAMILA uma vez que estas são naturalmente integradas no modelo calculado automaticamente a partir do DTD. Em termos práticos, cada restrição irá corresponder a um predicado que deverá sempre retornar o valor booleano *verdade*, caso contrário uma mensagem de erro terá de ser enviada ao utilizador.

Na nossa implementação, as restrições são especificadas por um conjunto de regras; cada regra, é um par formado por uma condição (a negação da respectiva restrição) e a reacção que lhe fica associada.

Os casos práticos que temos em mãos são de alguma complexidade e têm problemas muito específicos, o que os torna impróprios para exemplificar o nosso sistema. Assim, vamos recorrer a um exemplo mais pequeno, com complexidade suficiente para ilustrar as nossas ideias e o nosso protótipo.

Exemplo 8-2. Reis e Decretos

Um documento é composto por uma lista de decretos proclamados por um determinado rei.

Apresenta-se a seguir o respectivo DTD:

```
          <!DOCTYPE  rei  [
          <!ELEMENT  rei    -
(nome, cognome, datan, datam,decreto+)>
          <!ELEMENT  decreto - (data, corpo)>
          <!ELEMENT  (nome,cognome,datan,datam,data) -
(#PCDATA)>

          <!ATTLIST  (datan,datam,data)
                  valor CDATA #IMPLIED
                  tipo  CDATA #FIXED date>

          <!ELEMENT  corpo   - (#PCDATA)>
          ]>
```

Repare-se na declaração dos atributos *tipo* e *valor* para os elementos relacionados com datas. O atributo *tipo* tem um valor fixo que será usado mais tarde pelo processador de restrições (o validador semântico) para inferir o tipo do seu conteúdo. O atributo *valor* será utilizado para guardar a forma normalizada das datas; o processador usará o valor deste atributo, sempre que este estiver instanciado, em vez do conteúdo do elemento.

O texto seguinte é uma instância do referido documento, escrita de acordo com o DTD.

```
          <rei>
```

```
            <nome>D.Dinis</nome>
            <cognome>O Lavrador</cognome>
            <datan valor="1270.09.23»23 de Setem-
bro de 1270</datan>
            <datam valor="1370.09.23" >23 de Setem-
bro de 1370</datam>
            <decreto>
              <data valor="1300.07.15" >Ao décimo quin-
to dia do mês de Agosto
                  do ano 1300:</data>
              <corpo>A partir do dia de hoje, apenas bicicle-
tas poderão
                     circular na cidade de Braga.</corpo>
            </decreto>
            <decreto>
              <data>1389.11.03</data>
              <corpo>O McDonalds passará a vender vinho ver-
de em vez de COCA-COLA.</corpo>
            </decreto>
          </rei>
```

Observando o DTD e a respectiva instância podemos identificar de imediato algumas condições que devem ser verificadas:

- A *data* de cada decreto deverá sempre estar compreendida entre a data de nascimento (*datan*) e a data de falecimento (*datam*) do respectivo rei.
- O nome do rei deverá constar da nossa base de dados de pessoas famosas.

Para adicionarmos estas condições de contexto ao nosso sistema de validação, adicionamos ao DTD a declaração de entidades externas onde as condições estarão especificadas. Este método para associar restrições ao DTD representa apenas uma das três opções possíveis para o fazer e que são discutidas mais à frente (Secção 8.3).

```
          <!DOCTYPE rei [
          <!NOTATION CAM SYSTEM "camila.exe»
          <ELEMENT rei - -
(nome,cognome,datan,datam,decreto+)>
          <ENTITY  rei-rest SYSTEM "rei.cam" NDATA CAM>
          ...
```

```
          <ELEMENT decreto - - (...)>
          <ENTITY  decreto-
rest SYSTEM "decreto.cam" NDATA CAM>
          ]>
```

Neste caso, utilizamos uma entidade para cada conjunto de restrições.

No contexto do nosso exemplo, poderíamos escrever as seguintes restrições (utilizando a linguagem CAMILA):

```
rei(r) =
  { if(nome_(r) notin PessFamDB -
> nome_(r) ++ "Não existe..."),
    if(datan_(r) > datam_(r) -
> nome_(r) ++ "Morreu antes de nascer."),
    if(datam_(r) - datan(r) > 120 -
> nome_(r) ++ "Viveu demais!"),
    if(!all( x<-
decreto_l(r) : datan_(r) < data_(x) /\ data_(x) < datam_(r))
            -> nome_(r) ++ "fez um decreto fora da sua vida!")
  };
```

Se o nosso modelo instanciado tivesse os seguintes valores:

- `r = ("D.Dinis", "O Lavra-
  dor", "1265.06.24", "1211.04.12", ...)`
- `datan_(r) = "1265.06.24"`
- `datam_(r) = "1211.04.12"`

Este conjunto de valores faria disparar a regra

```
if(datan_(r) > datam_(r)
```

que produziria como resultado a concatenação ("++") do nome do rei ("nome_(r)") com a string "Morreu antes de nascer.".

8.3. Associação de Restrições aos Elementos

Tendo decidido sobre a linguagem que irá ser utilizada na escrita de restrições é altura de pensar na associação destas aos elementos que compõem o respectivo DTD.

A tarefa de escrever as restrições é atribuída ao analista que as deverá definir quando estiver também a definir o DTD.

A primeira decisão que foi tomada surge neste ponto: Como incluir as restrições no DTD sem alterar a sintaxe do SGML? Um dos objectivos é a não alteração do método de trabalho existente.

Analisando os construtores do SGML, pensou-se em três métodos alternativos:

secções de comentários especiais

Um comentário pode aparecer em qualquer lugar do DTD. Assim, podemos definir as restrições em comentários espalhados pelo DTD. Estes comentários levam uma marca para os distinguir dos comentários normais: a palavra reservada *Constraints* é utilizada.

Exemplo 8-3. Restrições como comentários

```
          <!DOCTYPE rei [
          <!ELEMENT rei - -
(nome,cognome,datan,datam,decreto+)>
          <!- Constraints
                   rei(k) = ...
          ->
          ...
          ]>
```

---

ficheiro externo

A ideia é utilizar de novo um comentário especial que indicará qual o ficheiro que tem as restrições que deverão ser associadas com o DTD corrente.

Exemplo 8-4. Restrições num ficheiro externo

```
<!DOCTYPE rei [
<!- Constraints: rei.cam ->
...
]>
```

---

entidade externa

Uma vez que os comentários são completamente ignorados pelas aplicações SGML, pode parecer estranho estarmos a utilizá-los para adicionar informação semântica aos documentos. Para quem possa pensar assim, avançámos com uma terceira alternativa bem dentro do SGML: a utilização de uma ou várias entidades externas.

Uma entidade é vista pelas aplicações como um componente do documento. No caso da validação, o parser irá tentar analisar o seu conteúdo. Neste caso, não nos interessa que isso aconteça uma vez que o conteúdo não é SGML. Para evitar essa situação, utilizámos a mesma técnica que se usa para imagens: declaramos que a entidade tem um conteúdo que deve ser tratado por uma aplicação externa que saberá processar a informação que estiver lá dentro.

Exemplo 8-5. Restrições numa entidade externa

```
          <!DOCTYPE rei [
          <!NOTATION CAM SYSTEM "camila.exe»
          <!ELEMENT rei - -
(nome,cognome,datan,datam,decreto+)>
          <!ENTITY  king-rest SYSTEM "king.cam" NDATA CAM>
          ...
          ]>
```

Na segunda linha, declara-se que CAM é uma notação que é interpretada pela aplicação "camila.exe". Depois, podemos indicar ao sistema que as entidades externas que contêm restrições têm o conteúdo nesta notação (linha 4).

Qualquer uma das três alternativas é válida e não implica qualquer alteração nos sistemas existentes em funcionamento.

Para o desenvolvimento dos casos de estudo escolhemos a segunda alternativa. A primeira mistura as restrições com as declarações do DTD e no caso de DTDs complexos faria com que estes ficassem ainda mais complicados de ler e de entender. A segunda e a terceira alternativas são funcionalmente equivalentes; a segunda é mais fácil de utilizar em prototipagem (os comentários são livres, as aplicações SGML não lhes atribuem qualquer semântica).

A separação das restrições do DTD vem ainda permitir a construção de uma pequena e útil ferramenta: um processador simples que analisa o DTD e produz o esqueleto do código das restrições. Desta maneira, o analista poderá trabalhar mais rápido e com mais segurança (no esqueleto já estão todos os protótipos das restrições, nenhum foi esquecido).

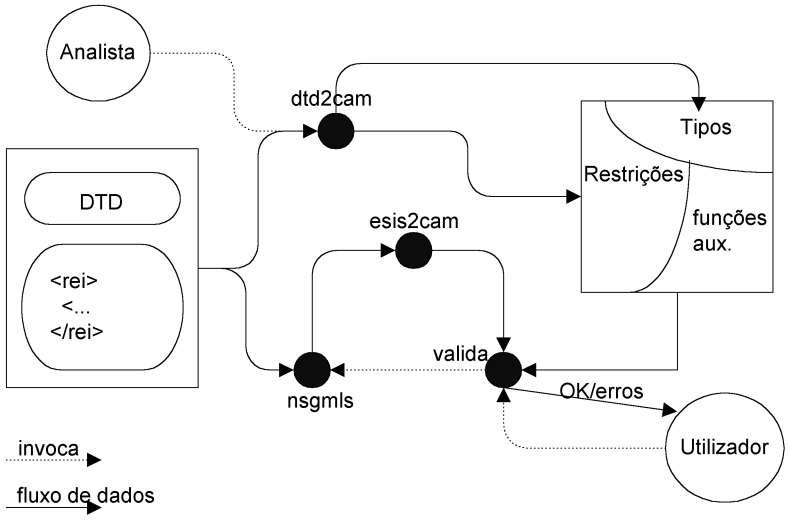
8.4. Processamento das Restrições

Depois de discutirmos a inclusão/ligação de restrições aos DTDs, vamos ver agora de que modo é que podemos acoplar o módulo de validação extra a um sistema existente.

Como foi apresentado na Figura 8-1, adicionou-se um processo extra ao modelo de processamento baseado em SGML tradicional. Este novo processo será o responsável pela execuçaõ das tarefas necessárias à verificação das restrições.

Na figura seguinte (Figura 8-2), podemos ver este processo com mais detalhe.

Figura 8-2. O Novo Componente de Validação – CAMILA



A adição deste novo processo exige adaptações a dois níveis: um relacionado com as pessoas intervenientes, analista e utilizadores, e outro relacionado com o software, como é que vão interactuar as novas rotinas com as já existentes. O primeiro nível será designado por *operacional* e o segundo por *aplicacional* e podem ser descritos da seguinte forma:

Nível Operacional

Neste nível consideramos que há sempre a intervenção de dois tipos de agente: o analista e o autor.

No modelo tradicional (Figura 7-1), o analista apenas tinha que desenvolver o DTD. No novo modelo, além do DTD, o analista terá de desenvolver ao mesmo tempo a especificação de restrições.

No fim da fase de análise, uma cópia do DTD é enviada ao processo de edição e, a mesma cópia juntamente com as restrições, é enviada a um novo processo de software que realizará a validação adicional (Figura 8-2).

Estas alterações são invisíveis para o utilizador do sistema. Ele só sentirá mudanças na preocupação que agora terá que ter em relação à normalização de

informação.

Nível de Aplicacional

Este nível diz respeito ao novo componente (o processo adicional de validação) que passaremos a descrever. Como se pode ver na Figura 8-2, este processo é composto por vários subcomponentes/funções. O maior e mais complexo foi designado por *Validador-CAMILA* (devido à linguagem e sistema de prototipagem em que foi desenvolvido - [ABNO97]). Os outros foram baptizados de acordo com a sua funcionalidade: *dtd2cam*, *esis2cam*, e *nsgmls* que é o conhecido parser de SGML desenvolvido por James Clark [SP].

O processo é centrado na função *Validador-CAMILA*, que recebe três tipos de informação provenientes dos outros intervenientes e produz um resultado. O DTD concebido na fase de análise é enviado ao *dtd2cam*, que traduz a definição dos elementos com as restrições associadas num modelo CAMILA (tipos com invariantes associados), passando este ao *Validador-CAMILA*. As restrições que são escritas directamente na linguagem CAMILA vão directamente para o *Validador-CAMILA*.

Do outro lado, e já com um autor a editar documentos, quando este quer validar o documento que está a editar, activa a função *valida* que começa por enviar o documento ao *nsgmls*; o output do *nsgmls* (o documento no formato ESIS - formato intermédio usado nas aplicações SGML), que normalmente é ignorado a menos das mensagens de erro, é enviado ao processo *esis2cam*, que vai usá-lo para instanciar a estrutura já definida em CAMILA, traduzindo as instâncias dos elementos no documento em expressões CAMILA (termos dos tipos de dados algébricos).

Assim, o *Validador-CAMILA* em posse do DTD (traduzido pelo *dtd2cam*), das restrições e do documento (traduzido pelos *nsgmls* e *esis2cam*) pode desencadear a validação semântica, bastando para isso executar as restrições.

Como resultado, o *Validador-CAMILA* envia ao utilizador um OK ou uma lista de mensagens de erro.

Podemos aperceber-nos facilmente que este processo não é simples e de que várias decisões foram tomadas no curso da sua implementação.

No resto do capítulo, acompanharemos o percurso dum documento que irá atravessar o nosso sistema. Aproveitaremos vários estados dessa passagem para ilustrar as decisões tomadas na sua implementação e respectivos porquês.

8.4.1. Implementação

O processo, cujo comportamento foi descrito na secção anterior, compreende duas tarefas de tradução.

A primeira é resolvida por um processador, *esis2cam*, que converte um documento, depois deste ter passado o processo de validação tradicional, em formato ESIS (Secção 5.3) em expressões CAMILA.

Aplicando o nsgmls ao nosso exemplo, o resultado ESIS seria o seguinte:

```
(rei
(nome
-D. Dinis
)nome
...
(decreto
A valor CDATA "1300.07.15"
A tipo  CDATA "date"
(data
-1300.07.15
)data
(corpo
-A partir do dia de hoje ...
)corpo
)decreto
...
)rei
```

No passo seguinte, **esis2cam** irá converter este resultado em termos do modelo abstracto CAMILA gerado por **dtd2cam**.

A segunda tarefa de tradução (**dtd2cam**) não é tão simples pois envolve um mapeamento entre um DTD e uma álgebra:

- o DTD é traduzido para um modelo na teoria de SETs.

- cada elemento é mapeado num tipo, de acordo com um esquema de tradução [RAH95, RAH98].
- cada elemento tem uma restrição a ele associada que ficará agora associada ao tipo.
- uma restrição é um conjunto de pares: condição - reacção; por defeito tem o valor verdadeiro; a condição e a reacção são escritas de acordo com a sintaxe CAMILA com a utilização de operadores CAMILA (Secção 2.1).

O próximo exemplo demonstra a aplicação deste processo ao caso dos "reis e decretos"que tem vindo a ser seguido.

Exemplo 8-6. Dum DTD para CAMILA

O DTD que temos vindo a usar é composto pelas seguintes declarações:

```
<!ELEMENT  rei    - (nome, cognome, datan, datam,decreto+)>
          <!ELEMENT  decreto - (data, corpo)>
          <!ELEMENT  (nome,cognome,datan,datam,data) -
(#PCDATA)>

          <!ATTLIST  (datan,datam,data)
                  valor CDATA #IMPLIED
                  tipo  CDATA #FIXED date>

          <!ELEMENT  corpo   - (#PCDATA)>
```

Aplicando **dtd2cam** a este DTD obtemos:

```
TYPE
  rei=nome_     :text
      cognome_  :text
      datan_    :date
      datam_    :date
      decreto_l :decreto-seq

  decreto=data_  :date
          corpo_ :text
ENDTYPE
```

```
rei(r)=true;

decreto(d)=true;

...
```

- Elementos estruturados são mapeados em tuplos: rei e decreto.
- Elementos atómicos sem um tipo associado são automaticamente definidos como sendo text: nome e cognome.
- Elementos com um tipo associado são definidos como sendo desse tipo: datan, datam e data.
- Elementos com indicadores de ocorrência são definidos como listas.
- Por fim, é gerado um invariante para cada elemento. O invariante gerado é uma tautologia podendo e devendo ser reescrito pelo analista conforme as restrições que este quiser impôr.

Termina aqui a travessia do nosso sistema.

Os exemplos práticos obtidos com esta abordagem demonstraram-nos que esta é uma boa solução para a implementação da validação semântica. Não aumenta muito a complexidade do sistema e não altera as aplicações SGML existentes.

Capítulo 9. Validação Semântica com Gramáticas de Atributos

Neste capítulo vamos apresentar uma solução baseada em gramáticas de atributos para o problema da definição e processamento de restrições semânticas.

Quando nos debruçamos mais sobre a semelhança destes dois mundos surgiu a ideia de se utilizar uma ferramenta de geração automática de compiladores para gerarmos editores SGML específicos. A ideia evoluiu mais um pouco e o resultado final reflecte-se no sistema S4.

O sistema S4 é um ambiente integrado para a programação de documentos. Uma das facilidades integradas no sistema é a possibilidade de definição e processamento de restrições (o problema que temos vindo a perseguir). Mas além desta, inclui outras como a integração duma sintaxe mais ligeira que o DSSSL para a especificação de estilos. O produto final pode resumir-se dizendo que é um gerador de editores estruturados baseados em SGML com possibilidade de definir e processar restrições semânticas e com a possibilidade de acoplar uma especificação de estilo que fará com que o editor gerado produza automaticamente vistas do documento com o conteúdo transformado, ou formatado.

Nas próximas secções e após uma introdução teórica que suporta a abordagem seguida, apresentaremos o sistema S4 e discutiremos os vários pormenores da sua implementação.

9.1. Aproximação via gramáticas de atributos

As operações que normalmente se fazem com documentos incluem: tradução, formatação do conteúdo, interpretação, acesso e recuperação de partes relevantes da informação contida no documento. Olhando para esta lista de operações, mais uma vez, o paralelismo entre o processamento de documentos e o processamento de linguagens formais emerge (ver o que se disse atrás na (Secção 6.1):

- para todas estas operações é necessário um primeiro passo de reconhecimento e

criação de uma representação intermédia - regra geral corresponde a uma análise léxica e uma análise sintáctica; no caso dos documentos SGML são realizadas pelo parser genérico de SGML; no caso das linguagens é também realizado pelo parser específico de cada linguagem.

- a interpretação corresponde a uma análise semântica que é feita após as duas primeiras.
- a tradução e a formatação correspondem a travessias da representação intermédia - quer num caso quer no outro têm de ser programadas de acordo com o objectivo final pretendido.
- o acesso e recuperação de partes correspondem a travessias da representação intermédia com filtragem de componentes - no caso dos documentos SGML corresponde à realização de uma query; nas linguagens, em termos de funcionalidade, corresponderá parcialmente à optimização do código gerado.

Nesta secção, vamos aplicar ao processamento documental uma técnica que já se vem aplicando no processamento de algumas linguagens formais. Nesta abordagem representaremos a semântica dum documento como uma *Árvore de Sintaxe Abstracta Decorada (ASAD)*. Uma ASAD é formalmente especificada por uma gramática de atributos. Relativamente à semântica, vamos deixar de fora na da introdução as restrições de contexto que se poderão associar a um documento SGML e que temos vindo a discutir em capítulos anteriores. O objectivo aqui é verificar se esta aproximação tem capacidade para representar os documentos SGML com todas as restrições possíveis de expressar num DTD e que um sistema SGML tradicional automaticamente valida. Só depois iremos discutir se é ou não possível acrescentar restrições semânticas extra SGML mas relacionadas com o documento em causa.

A notação relativa às gramáticas de atributos que irá ser usada ao longo deste capítulo, nas especificações sintácticas e semânticas dos exemplos, foi já devidamente apresentada no capítulo introdutório (Secção 2.2). Nas partes mais específicas e relacionadas com a implementação, é utilizada a linguagem SSL - "Synthesizer Specification Language", também introduzida numa secção do capítulo inicial (Secção 2.2.2).

O primeiro passo para a construção desta representação semântica de documentos é a concepção da *Gramática Independente de Contexto (GIC)* que lhe servirá de base. É sempre possível derivar uma GIC sistematicamente, ou automaticamente,

partindo do DTD. A tabela seguinte, Tabela 9-1, apresenta um primeiro esboço dum esquema de tradução entre os dois formalismos.

Tabela 9-1. Esquema de Tradução SGML ↔ Gramáticas de Atributos

SGML	Gramáticas de Atributos
x,y	$Z \rightarrow X\ Y$
x & y	$Z \rightarrow X\ Y \mid Y\ X$
x | y	$Z \rightarrow X \mid Y$
x*	$Z \rightarrow X$; $X \rightarrow$ X-elem X $\mid \epsilon$
x+	$Z \rightarrow X$; $X \rightarrow$ X-elem X | X-elem
x?	$Z \rightarrow X \mid \epsilon$

A GIC que se segue foi obtida sistematicamente a partir do DTD apresentado para o tipo de documentos *literate programming* (Exemplo 8-1).

Exemplo 9-1. GIC derivada do DTD de Literate Programming

```
p1:     litprog  -> "<litprog>" X "</litprog>"
p1.1:   X        -> TEXTO X | prog X | def X
p1.2:               | id X | sec X | tit X
p1.3:               |

p2:     prog     -> "<prog>" Y "</prog>"
p2.1:   Y        -> TEXTO Y | id Y
p2.2:               |

p3:  def      -> "<def ident=" ID »" prog "</def>"
p4:  sec      -> "<sec>" TEXTO "</sec>"
p5:  tit      -> "<tit>" TEXTO "</tit>"
p6:  id       -> "<id refid=" ID »"
```

Depois de se obter a GIC, o passo seguinte consiste na associação de atributos aos símbolos da GIC e na escrita das respectivas regras de cálculo (no contexto de cada regra de derivação).

A introdução de atributos vai permitir duas coisas: por um lado, expressar as condições de contexto necessárias para restringir o conjunto de frases válidas (predicados que têm de ser verificados no contexto de algumas produções); por outro lado, transportar através dos atributos toda a restante informação não estrutural que é inferida do documento, directa ou indirectamente, e que é necessária para realizar a transformação do documento.

No primeiro caso, diremos que o objectivo é a definição da semântica estática e, no segundo, a definição da semântica dinâmica.

Os atributos correspondentes à semântica estática são facilmente deriváveis a partir das cláusulas *ATTLIST* presentes no DTD.

Considerando de novo o nosso caso de estudo, a derivação dos seguintes atributos (associados aos símbolos **def** e **id**) é imediata:

```
def: syn {ident: word}

id:  syn {refid: word}
```

Aos quais associamos as seguintes regras de cálculo:

```
p3:  def  -> "<def ident=" ID »" prog "</def>"
       ident(def) = lexval(ID)

p6:  id   -> "<id refid=" ID »"
       refid(id) = lexval(ID)
```

Assumiu-se que cada símbolo terminal (no nosso exemplo ID e TEXTO) tem um atributo intrínseco chamado *lexval* (valor léxico) do tipo *word*.

A única restrição de contexto que é preciso especificar, para este exemplo, em termos de atributos diz respeito à validação de atributos do tipo ID e IDREF que um parser SGML faz (é das poucas validações semânticas realizadas): cada ID só aparece definido uma vez e cada IDREF deve ter um valor de um ID definido algures no documento. Assim, no nosso caso, definimos as seguintes condições de contexto:

```
p3:  def      -> "<def ident=" ID »" prog "</def>"
CC:    not exists( ident(def), itab(def) )
```

```
p6:  id       -> "<id refid=" ID »"
CC:    exists( refid(id), itab(id) )
```

Para escrever estas condições de contexto, vimo-nos perante a necessidade de associar aos símbolos *def* e *id* um atributo herdado, *itab*. Este atributo faz o papel de uma tabela de símbolos, que aqui vai registando os identificadores que são definidos ao longo do documento, e que é consultada sempre que um identificador é referenciado. Para manter esta tabela actualizada é ainda necessário acrescentar os seguintes atributos, cuja semântica é explicada mais à frente:

```
def:  inh {itab: IdentTAB}
      syn {pros: PEleList}

id:   inh {itab: IdentTAB}

X:    inh {itab: IdentTAB}
      syn {stab: IdentTAB}

prog: syn {pros: PEleList}
```

Com as seguintes regras de cálculo:

```
p3:  def      -> "<def ident=" ID »" prog "</def>"
       pros(def) = pros(prog)

     X        -> def X
       itab(def) = itab(X0)
       itab(X1)  = actuali-
za( itab(X0), ident(def), pros(def) )
       stab(X0)  = stab(X1)


     X        -> id X
       itab(id) = itab(X0)
       itab(X1) = itab(X0)
       stab(X0) = stab(X1)
```

O desenvolvimento da gramática de atributos prosseguiria desta forma até todas as restrições semânticas estarem cobertas e todos atributos utilizados nessas restrições estarem especificados.

Para completar o nosso exemplo temos de definir ainda os atributos *pros* e *tab* associados ao axioma da GIC, *litprog*; *pros* será uma estrutura associativa onde serão guardados pares com a seguinte composição: **identificador–>pedaço-de-programa**; *tab*, tal como os acima referidos *itab* e *stab*, é uma tabela de identificadores normal.

```
litprog: syn {tab: IdentTAB; pros: PEleList}

X:       syn {pros: PEleList}
```

E as respectivas regras de cálculo:

```
X  -> prog X
  pros(X0) = append( pros(prog), pros(X1) )

litprog -> "<litprog>" X "</litprog>"
  tab(litprog) = stab(X)
  pros(litprog) = pros(X)
```

Neste ponto, dispomos do necessário para especificar qualquer processador para este tipo de documentos via equações/funções sobre os atributos. Por exemplo, um extractor de programas poderia ser especificado da seguinte maneira:

```
p1:  litprog -> "<litprog>" X "</litprog>"
       getprogram( pros(litprog), tab(litprog) )
```

O procedimento *getprogram* vai percorrer as duas estruturas criadas durante o reconhecimento do documento, respectivamente *pros* que tem o código dos excertos de programas e *tab* que tem a informação sobre os identificadores.

Com a análise deste exemplo pretende-se mostrar a capacidade deste formalismo para representar a semântica em documentos. Pode-se, então, concluir que, não só tem capacidades para o fazer, como nos dá uma margem de manobra para criar extensões (a linguagem de cálculo de atributos é um novo universo operacional). No exemplo seguido, fez-se uma conversão manual dum DTD para uma gramática mas,

se considerarmos a tabela de conversão construída (Tabela 9-1) é fácil ver que este processo é sistemático e por isso automatizável. E é este o objectivo que iremos perseguir ao longo deste capítulo, automatizar o processo de modo a que as gramáticas de atributos não passem de uma mera curiosidade científica para o utilizador.

Na próxima secção (Secção 9.2) iremos descrever o desenvolvimento e implementação dum sistema que irá automatizar muitas das fases do ciclo de vida dos documentos SGML.

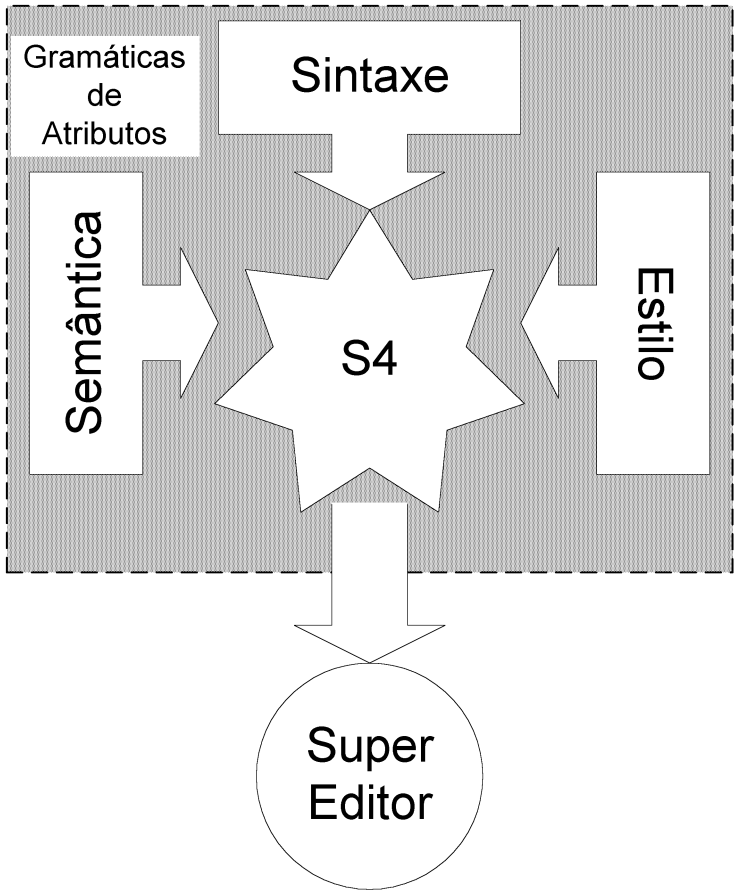
Os conceitos fundamentais relacionados com o sistema S4 foram apresentados ao longo dos capítulos anteriores: O SGML como objectivo de edição, as gramáticas de atributos como fundamento de especificação, e o *Synthesizer Generator* como primeira ferramenta de implementação, tendo esta implementação evoluído recentemente para o sistema LRC.

9.2. O sistema S4

Apesar de ter sido um passo em frente na caótica produção documental, o SGML não é de todo acessível ao utilizador comum. Quando, numa determinada linha administrativa, se faz a opção de produzir toda a documentação em SGML, algum cuidado deve ser colocado na adaptação dos utilizadores a esta nova maneira de "pensar"e "produzir"documentos. Esta foi uma das preocupações que esteve na génese do S4. A constatação do que se passa com as páginas HTML veio também reforçar a ideia de criar o S4: não é preciso ser-se um utilizador muito experiente para conseguir produzir documentos para disponibilizar no "*World Wide Web (WWW)*"; muita gente, nas mais variadas áreas profissionais, produz páginas HTML; na maior parte das vezes, estes utilizadores não têm consciência de que aquilo que estão a fazer é escrever um documento SGML de acordo com um DTD - o DTD do HTML. É, precisamente, esta transparência e simplicidade de utilização que nos interessa captar para o sistema S4.

O nome S4 vem do inglês *"Syntax, Semantics and Style Specification"*.

Figura 9-1. S4 - os conceitos



O S4 surge como a reunião de todos os problemas e soluções apontados e desenvolvidos no decorrer desta tese. A ideia, esquematizada na Figura 9-1, é juntar debaixo do mesmo chapéu as três grandes linhas da edição estruturada de documentos: a sintaxe, a semântica e o estilo.

No fim, o sistema resultante será capaz de gerar editores estruturados WYSIWYG sensíveis a restrições semânticas. O ponto comum, o chapéu de que falávamos há pouco, é um sistema gerador de compiladores baseado em gramáticas de atributos; inicialmente começou por ser o "*Synthesizer Generator*"[RT89a, RT89b], mas, a breve prazo, será iniciado um projecto de implementação em LRC [SAS99] ou ANTLR [ANTLR] - a ideia é encontrar o ambiente com maior capacidade de

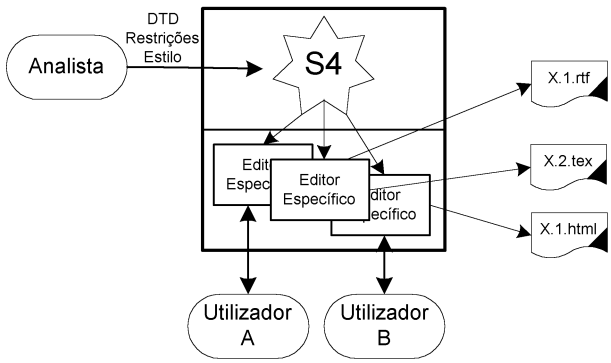
portabilidade nos editores gerados, o SGen é bastante limitado, o LRC e o ANTLR são menos limitados mas perdem algumas características de qualidade que o outro tem.

Na próxima secção (Secção 9.2.1), descreve-se a arquitectura funcional do sistema, do ponto de vista do utilizador e do analista, antes de se apresentar a descrição do interior do sistema. Na secção seguinte (Secção 9.2.2), descreve-se o suporte da linguagem de restrições, um dos contributos específicos desta tese. Por fim mostra-se a aplicação da linguagem aos problemas dos casos de estudo levantados ao longo da tese.

9.2.1. Arquitectura do S4

Na Figura 9-2, apresenta-se a arquitectura básica de utilização do sistema.

Figura 9-2. S4 - Ambiente para Programação de Documentos



O sistema disponibiliza duas interfaces diferentes: uma para o utilizador e outra para o analista. A ideia é ter um sistema integrado com dois níveis de acesso diferentes. Um de carácter administrativo e outro de utilização.

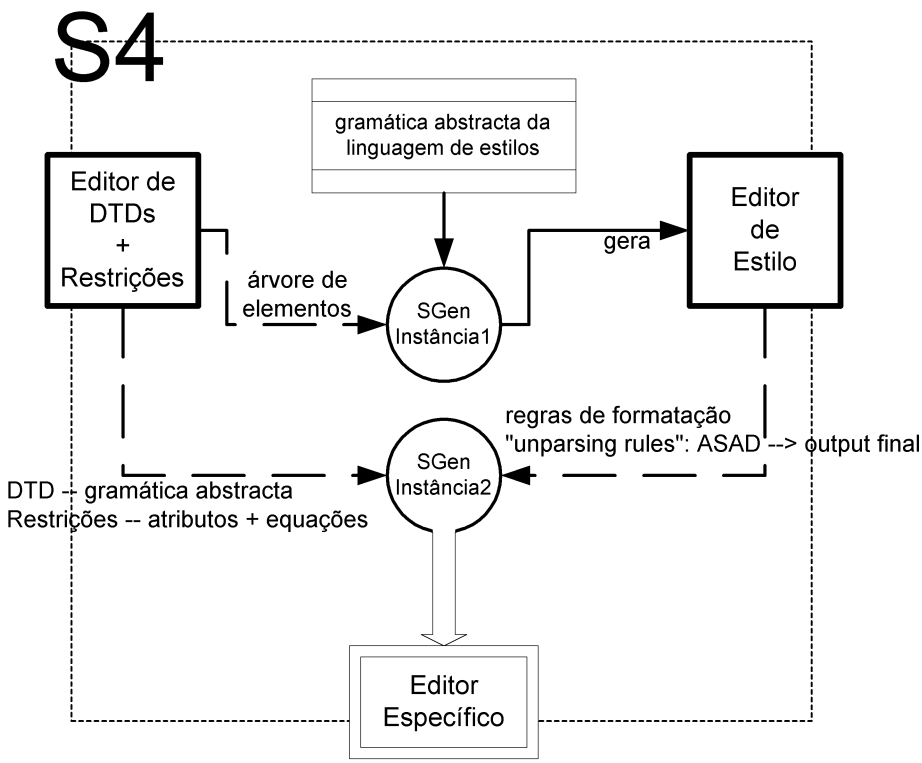
A interface disponibilizada ao analista permite-lhe definir novos tipos de documento: qual a sua estrutura, quais os invariantes sobre o seu conteúdo que deverão ser verificados e qual a aparência/estilo (uma ou mais) final dos documentos desse tipo.

O utilizador apenas se servirá do sistema para produzir os seus documentos por isso

apenas lhe é permitida a escolha de um tipo de documento e dentro dos estilos disponíveis para esse tipo qual o pretendido.

Na Figura 9-3, apresenta-se o esquema da organização interna do S4, com especial relevo para as relações e dependências entre os vários editores.

Figura 9-3. S4 - Estrutura interna



No S4, os *botões da máquina de lavar* são os três editores representados na Figura 9-3. Como iremos ver a figura referida representa também uma espécie de fluxograma do S4.

O analista tem à sua disposição dois editores, de certa forma interligados, um editor sensível à sintaxe do SGML que foi enriquecido com sintaxe extra de modo a permitir especificar restrições e outro que é sensível a uma sintaxe muito semelhante à linguagem *"eXtended Style Language"* (XSL - [xsl]). O primeiro serve

para o analista especificar o DTD e as restrições semânticas, e o segundo para especificar o estilo, o aspecto visual de cada elemento.

O editor de DTDs é genérico, conhece a sintaxe SGML e permite ao analista especificar qualquer DTD. O editor de estilos já não é genérico mas específico e dependente do DTD definido no editor de DTDs. O editor de estilos é gerado pelo editor de DTDs e representa o primeiro resultado daquele. A ideia por detrás desta dependência é a de que, conhecendo a estrutura dum determinado tipo de documentos (DTD), a especificação de estilo será restrita aos elementos presentes nessa estrutura. Assim, pode-se gerar um esqueleto da especificação de estilo automaticamente partindo do DTD, com a vantagem de nenhum elemento ter sido esquecido.

Como já foi amplamente discutido, um DTD não é mais do que uma gramática (independente de contexto, se nos abstivermos de usar algumas das facilidades do SGML que uma vez que não foram implementadas no nosso editor não levantam qualquer problema) o que torna de certa maneira simples a geração da gramática abstracta correspondente (Secção 9.2.1.1). E é esta gramática abstracta, representada na sintaxe SSL, o segundo resultado do editor de DTDs.

O editor de estilo gerado pelo editor de DTDs é específico dum determinado DTD e vai permitir ao analista associar um estilo a cada um dos elementos nesse DTD (Secção 9.2.1.2).

Recebendo estas especificações (DTD, restrições, especificação de estilo) o S4 vai usar uma segunda instância do SGen para processá-las e produzir um novo editor estruturado. Este editor é específico e permitirá ao utilizador escrever documentos de um determinado tipo.

O cerne do S4, o editor de DTDs com restrições, foi também gerado da mesma maneira que os novos editores; especificou-se em SSL o SGML; adicionou-se a especificação da sub-linguagem de restrições; e, utilizou-se o SGen para compilar tudo e gerar o editor final. Desta maneira, o S4 encontra-se implementado usando uma filosofia de *"bootstrap"*.

9.2.1.1. Editor de DTDs com Restrições

O editor de DTDs com Restrições, na sua primeira versão, foi objecto de estudo dum projecto de investigação (DAVID - especificação e processamento algébrico de documentos, [HAR99]) e duma tese de mestrado, onde o autor da presente tese

participou como co-orientador. Remete-se, por isso, o leitor interessado nos pormenores de implementação para a tese já referida [Lop98].

Depois da descrição das origens e do modo como foi concebido, vamos ver o que este editor é capaz de fazer.

A tarefa mais relevante deste editor é a conversão dum DTD para uma gramática abstracta. Como já foi discutido, procurou-se automatizar este processo. A automatização foi conseguido graças à elaboração dum conjunto de regras que no seu todo formam o algoritmo de conversão. No Apêndice C, encontram-se descritas todas as regras que se sumarizam abaixo, pois serão importantes nas discussões que se seguirão:

Elemento do DTD → Símbolo não-terminal da gramática abstracta

Para cada elemento do DTD é gerado um símbolo não-terminal da gramática. O conjunto de produções deste símbolo permite derivar o conteúdo associado a este elemento. Estas produções são geradas num ou mais passos conforme a expressão de conteúdo do elemento: de notar que no DTD existem operadores de ocorrência que não têm equivalência na notação gramatical (há notações gramaticais com BNF estendido [ANTLR], mas o SSL não pertence a esta categoria); a conversão consegue fazer-se usando a técnica de conversão de expressões regulares em gramáticas regulares (desde que algumas facilidades doSGML, que podem originar linguagens ambíguas ou não-regulares, não estejam presentes; são exemplos dessas facilidades o operador &, as inclusões e as exclusões; nestes casos algoritmos específicos teriam de ser utilizados mas nem todas as situações poderiam ser resolvidas - [Bru94]).

Atributo dum Elemento do DTD → Símbolo não-terminal da gramática abstracta

Neste ponto e logo à partida, há um conflito de nomes que é necessário esclarecer: o atributo dum elemento dum DTD é diferente do atributo dum símbolo duma gramática de atributos. O primeiro é um subelemento estrutural do elemento a que pertence, e o segundo é utilizado para associar informação semântica a um símbolo. Podemos ver isto ainda da seguinte maneira, em SGML temos dois níveis de especificação estrutural, o elemento e o atributo, nas gramáticas temos apenas um nível, o símbolo. A existência destes dois níveis em SGML tem sido alvo de muitas discussões (Secção 5.1.6) e, muito recentemente, tem até sido posta em causa (Secção D.7) – o mais recente

movimento a favor duma *"Simplified Markup Language - SML"* defende, entre outras coisas a eliminação de atributos mediante a unificação destes com os elementos.

A solução para a eliminação dos atributos do SGML passa pela promoção daqueles a elementos (quem está habituado a usar SGML irá sempre arguir que algo se perdeu). Uma vez que nas gramáticas temos apenas um nível de especificação estrutural, para convertermos DTDs em gramaáticas vamos ter que colocar atributos e elementos no mesmo plano. Assim, atributo e elemento serão tratados da mesma maneira e um atributo, tal como o elemento corresponderá a um símbolo não-terminal.

Como há vários tipos de atributo, cada um com a sua semântica específica, aqui as suas derivações também serão diferentes: um atributo do tipo enumerado corresponderá a um símbolo não-terminal com uma produção para cada um dos possíveis valores; um atributo com valor por omissão #IMPLIED terá duas produções, uma vazia e outra para o respectivo valor; um atributo com valor por omissão #REQUIRED será representado directamente pelo seu valor; ...

Entidade do DTD → Símbolo não-terminal da gramática abstracta

No caso das entidades, guardamos o seu identificador e os seus atributos num atributo da gramática que implementa uma tabela de símbolos convencional [PP92] e que será associado aos vários símbolos gramaticais (gerados pelas regras anteriores).

O principal efeito da declaração duma entidade faz-se sentir nos elementos textuais, pois aquelas podem aparecer em qualquer ponto do texto. Assim, a derivação de um elemento de conteúdo textual tem que contemplar a possibilidade de no meio do texto aparecerem referências a entidades.

Ao longo desta secção, a questão das restrições e da linguagem usada para as especificar tem sido tratada de forma algo leviana. Isto deve-se ao facto de uma das próximas secções ser inteiramente dedicada a isso (Secção 9.2.2).

A título de exemplo, apresenta-se na subsecção seguinte a conversão de um DTD pertencente a um pequeno caso de estudo.

9.2.1.1.1. Um sistema noticioso automático

Este exemplo foi extraído de uma aplicação real desenvolvida no âmbito do projecto GEiRA [GEIRA]. Neste exercício, utilizou-se um servidor de email especial para implementar um serviço noticioso automático. Qualquer pessoa que queira reportar um facto, um evento, ..., envia uma mensagem de email que tem de obedecer a uma certa estrutura. Esta mensagem é processada pelo nosso servidor e se tudo estiver correcto a mensagem é publicada numa página WWW que actua como um boletim noticioso.

A estrutura da mensagem é definida pelo DTD a seguir listado.

Exemplo 9-2. O DTD da Notícia

O DTD que define o tipo *Notícia* é o seguinte:

```
<!DOCTYPE News [
<!- root element News has title, begin-date, an optional
     end-date and an optional body                          ->
<!- it also has two required attributes: type and subject ->

<!ELEMENT News - - (Title, Begin-date, End-date?, Body?)>
<!ATTLIST News Type (Event | Novelty) #REQUIRED
               Subject (Nature | Culture | Science) #REQUIRED>

<!ELEMENT Title      - - (#PCDATA)>
<!ELEMENT Begin-date - - (#PCDATA)>
<!ELEMENT End-date   - - (#PCDATA)>
<!ELEMENT Body       - - (Para)+>
<!ELEMENT Para       - - (#PCDATA)>
]>
```

De acordo com o nosso ponto de vista, uma mensagem que nos fosse enviada por este sistema, para ser tomada como válida, teria que respeitar o DTD e as seguintes restrições semânticas:

- As notícias do tipo *Event* têm de ter uma *Begin-date* (já exigida pelo DTD) e uma *End-date*.
- A *End-date* duma notícia do tipo *Event* deverá sempre ser maior ou igual que a *Begin-date* dessa notícia.

No nosso caso, expressamos estas restrições nos seguintes comentários especiais:

```
<!-CONSTRAINT: if News.type = "Event" then End-date != Null->

<!-CONSTRAINT:  if News.type = "Event" then Begin-date <= End-
date->
```

Aplicando as regras brevemente enunciadas no início desta secção e detalhadas em apêndice Apêndice C, convertemos este DTD na Gramática independente de contexto que se segue.

Exemplo 9-3. Gramática Independente de Contexto para a Notícia

```
News → NewsAttList NewsContent

NewsAttList → Type Subject
Type → EVENT
       | NOVELTY
Subject → NATURE
          | CULTURE
          | SCIENCE

NewsContent → Title Begin-date Grupo1_01 Grupo2_01
Title → TitleAttList TitleContent
TitleContent → textLst
TitleAttList → ϵ
Begin-date → Begin-dateAttList Begin-dateContent
Begin-dateAttList → ϵ
Begin-dateContent → textList

Grupo1_01 → Grupo1
          | ϵ
```

```
Grupo1 → End-date
End-date → End-dateAttList End-dateContent
End-dateAttList → ϵ
End-dateContent → textList

Grupo2_01 → Grupo2
          | ϵ

Grupo2 → Body
Body → BodyAttList BodyContent
BodyAttList → ϵ
BodyContent → Grupo3_1n

Grupo3_1n → Grupo3 Grupo3_1n
          | Grupo3


Grupo3 → Para
Para → ParaAttList ParaContent
ParaAttList → ϵ
ParaContent → textList

textList → text textList
         | ϵ
text → STR
```

É fácil ver que há muitas produções redundantes, mas elas são resultantes de um processo automático de conversão, que pode depois ser bem optimizado: por exemplo, elementos sem atributos não precisam de ter o terminal *ElemNameAttList* e as respectivas produções nulas.

Para completar o exemplo, falta fazer a conversão/derivação das restrições semânticas. Ao contrário do ponto anterior, em que as regras de conversão já se encontram formalmente especificadas, para a conversão das restrições semânticas vamos seguir algumas regras empíricas. O processo ainda está em fase de sistematização. A própria sintaxe em que se expressam as restrições ainda está em discussão como veremos na próxima secção (Secção 9.2.2).

Exemplo 9-4. Conversão da primeira restrição semântica da Notícia

A primeira restrição semântica exige que todas as notícias do tipo *Event* tenham uma data de fim (*End-date*) e foi definida assim:

```
<!-CONSTRAINT: if News.type = "Event" then End-date != Null->
```

A correspondente especificação gramatical é:

```
News → NewsAttList NewsContent
  CC:    if(NewsAttList.newsType = "Event")
         then if(NewsContent.endDate != )
              then "OK"
              else "ERRO"

NewsAttList → Type Subject
  NewsAttList.newsType = Type.newsType
Type → EVENT
  Type.newsType = "Event"
     | NOVELTY
  Type.newsType = "Novelty"

NewsContent → Title Begin-date Grupo1_01 Grupo2_01
  NewsContent.endDate = Grupo1_01.endDate

Grupo1_01 → Grupo1
  Grupo1_01.endDate = Grupo1.endDate
          | ϵ
  Grupo1_01.endDate =

Grupo1 → End-date
  Grupo1.endDate = End-date.endDate
...
End-dateContent → textList
  End-dateContent.endDate = Conv(date, textList)
```

newsType é o atributo que transportará ao longo da árvore de sintaxe o "tipo da notícia". O seu valor é sintetizado nas produções associadas a *Type*. Por sua vez, o atributo *endDate* é sintetizado nas porduções associadas ao NT *End-date* e é copiado ao longo da árvore até à raíz da subárvore a partir da qual se sintetiza este e o atributo *newsType*.

A segunda restrição é semelhante à primeira diferindo apenas no facto da restrição ser sobre o conteúdo de elementos e não sobre o valor de atributos.

Exemplo 9-5. Conversão da segunda restrição semântica da Notícia

A segunda restrição é um predicado que diz que se a Notícia é do tipo *Event* então a data de fim terá que ser superior à data de início, e foi definida assim:

```
<!-CONSTRAINT:  if News.type = "Event" then Begin-date <= End-
date->
```

A correspondente especificação gramatical é:

```
News → NewsAttList NewsContent
  CC:    if(NewsAttList.newsType = "Event")
         then if(NewsContent.beginDate <= NewsContent.endDate)
              then "OK"
              else "ERRO"

NewsAttList → Type Subject
  NewsAttList.newsType = Type.newsType
Type → EVENT
  Type.newsType = "Event"
     | NOVELTY
  Type.newsType = "Novelty"

NewsContent → Title Begin-date Grupo1_01 Grupo2_01
  NewsContent.beginDate = Begin-date.beginDate
  NewsContent.endDate = Grupo1_01.endDate

Grupo1_01 → Grupo1
  Grupo1_01.endDate = Grupo1.endDate
           | ϵ
  Grupo1_01.endDate =

Grupo1 → End-date
  Grupo1.endDate = End-date.endDate
...
```

```
End-dateContent → textList
  End-dateContent.endDate = Conv(date, textList)

Begin-dateContent → textList
  Begin-dateContent.beginDate = Conv(date, textList)
```

Neste caso, temos dois atributos, beginDate e endDate, cujos valores são sintetizados. O valor de beginDate é transportado ao longo da árvore até ao ponto onde endDate é sintetizado. Nesse ponto, em posse dos dois valores já é possível calcular a condição de contexto.

Ao longo desta secção, vimos como está implementado o editor de DTDs com Restrições na parte que diz respeito aos DTDs. A questão das Restrições por ser muito importante no contexto desta tese foi delegada para uma secção específica no fim do capítulo (Secção 8.2).

Na próxima secção, vamos "visitar"o editor de estilo que é muito semelhante ao que acabou de ser descrito.

9.2.1.2. Editor de Estilo

O editor de estilo é em tudo muito semelhante ao editor de DTDs com restrições:

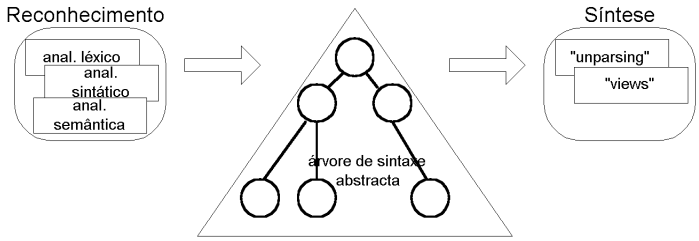
- A ideia da sua integração no sistema S4 surge também no âmbito do projecto DAVID.
- A sua concepção deu origem a uma tese de mestrado [Sou98], também co-orientada pelo autor da presente dissertação.
- Foi gerado pelo SGen a partir duma especificação SSL (gerada pelo editor de DTDs).
- Como resultado, gera parte da especificação SSL dum editor (para ser integrada com a outra parte que é gerada pelo editor de DTDs.

A filosofia e a integração destes editores estão relacionadas com a metodologia e a ferramenta que se estão a utilizar. Um editor estruturado gerado pelo SGen, ou pelo LRC, é composto por dois blocos funcionais: um faz o reconhecimento sintático e

semântico e como resultado produz uma representação intermédia (uma árvore de sintaxe abstracta decorada); o outro realiza uma tarefa de síntese, faz uma travessia à representação intermédia e gera normalmente uma visão transformada dos dados da árvore.

Na figura seguinte, apresenta-se este esquema.

Figura 9-4. Arquitectura funcional dum editor estruturado



O reconhecimento é essencialmente especificado pela gramática de atributos da linguagem que se está a reconhecer e usa equações sobre os atributos da gramática para construir a representação intermédia. A síntese é essencialmente especificada pelas *"unparsing rules"*, que também podem recorrer a equações sobre atributos para transformar parte da informação.

Sobre o editor de estilos, e uma vez que a sua concepção e funcionamento é muito semelhante ao editor de DTDs, não nos vamos alargar mais. O leitor mais interessado poderá recorrer à leitura da referida tese de mestrado onde é discutida a linguagem implementada e a concepção do editor.

O resto deste capítulo será inteiramente dedicado à linguagem de restrições: sua definição e implementação.

9.2.2. Linguagem de Restrições

Até este ponto, o leitor atento deverá ter notado que a preocupação com a linguagem de restrições está patente ao longo de quase toda a tese. A motivação por detrás dela e a própria linguagem em si são um dos principais contributos desta tese.

No entanto, é notória a obscuridade que paira sobre a definição da sua sintaxe. A ideia seguida foi a de discutir este assunto ao nível mais abstracto possível tentando fugir da sua concretização.

Mesmo na abordagem com modelos abstractos seguida no capítulo anterior (Capítulo 8), fugiu-se a esta questão. Utilizou-se a linguagem do ambiente de prototipagem para especificar as restrições. Pode parecer que a solução adoptada então é semelhante à que se defende mais à frente neste capítulo. Há, no entanto, uma grande diferença. A linguagem utilizada para a especificação de restrições não tinha nenhuma relação com o SGML ou sequer tinha sido pensada para o processamento de documentos estruturados, o que levou a que o documento SGML fosse processado por uma ferramenta (análise sintática) e as restrições por outra (análise semântica). A solução que se persegue aqui irá no sentido de integrar estes dois processamentos. Como se poderá ver mais à frente, a sintaxe que se irá escolher para a especificação das restrições está muito ligada à natureza dos documentos estruturados SGML, tornando mais fácil a sua percepção e aquisição.

Nesta altura, já não é possível fugir mais à sua definição pois estamos a discutir a implementação dum sistema em que ela aparece integrada. E, embora os editores gerados pelo Sgen (usado nas primeiras versões do S4) não necessitem duma sintaxe concreta para arrancar e funcionar, vão precisar dela sempre que se tentar reutilizar algum texto criado pelo sistema.

Em termos abstractos, e supondo que as restrições sejam simplesmente locais a um elemento (condições associadas sempre ao elemento), basta-nos enriquecer a definição do elemento da seguinte maneira:

```
elem : Element( identificador, expressão-conteúdo, restrição);
```

Um elemento passa a ser um triplo composto por um identificador, uma expressão de conteúdo e uma restrição (simples ou composta). Faltaria agora continuar a definir (em termos abstractos) a estrutura de cada um dos elementos do triplo.

Por outro lado, a sintaxe concreta que queremos ter no nosso sistema é a do SGML e já vimos quais as possibilidades de adicionar restrições usando a sintaxe SGML, sem a alterar (Secção 8.3). Qualquer uma das soluções apontadas não prevê uma associação implícita, duma restrição a um elemento. Essa associação tem de ser feita e a solução mais simples é usar a técnica das bases de dados para relacionar duas tabelas: neste caso, usar um dos campos do triplo.

O campo mais lógico para ser utilizado como chave da relação entre a definição do elemento e da restrição é o identificador do elemento. Porém a realidade é diferente da suposição feita acima e na prática levanta-se outro problema: o mesmo elemento pode aparecer em contextos diferentes e por isso, ter semânticas diferentes. Portanto as restrições não devem ser associadas a um elemento mas sim a um contexto, donde o que precisamos não é do identificador do elemento mas de algo ligeiramente mais complexo: um selector de contexto.

Assim, se usarmos comentários especiais como sintaxe de base para as restrições, podíamos ter uma sintaxe concreta com a seguinte forma:

```
<!- CONSTRAINT sel-contexto (restrição) ->
```

Ou seja, há duas coisas a considerar numa restrição, a selecção do contexto, e a especificação da restrição propriamente dita.

Este ponto, permitiu abrir a discussão da sintaxe concreta da linguagem de restrições e dividi-la desde já em duas partes: a sub-linguagem de selecção de elementos e a sub-linguagem com a qual se irão especificar as restrições.

A operação de seleccionar os elementos com os quais se quer fazer alguma coisa, ou aos quais se quer aplicar algum processamento, tem sido, desde há algum tempo, uma preocupação das pessoas que trabalham nesta área. Começou por surgir na transformação e na formatação: era preciso seleccionar os elementos que se queriam transformar, ou que se queriam mapear num ou mais objectos com características gráficas (formatação). Este esforço é visível no DSSSL (já discutido na Secção 6.2); o primeiro elemento das suas regras é uma expressão de *"query"* que selecciona os elementos aos quais será aplicado o processamento especificado. Por último, esta necessidade surgiu ligada às linguagens de *"query"* para documentos estruturados, como as que foram propostas na última conferência dedicada a esse tópico [RLS98, DeR98, Tom98, Wid8, CCDFPT98].

Até aqui só se levantou a ponta do véu e já começaram a proliferar linguagens, todas elas com o mesmo objectivo. Foi o que constataram alguns investigadores ligados à área e que começaram a olhar com mais atenção para cada uma delas tentando extrair um denominador comum.

Assim se chegou, rapidamente, à conclusão de que a operação de selecção necessária para a transformação ou formatação era muito semelhante à necessária nos sistemas de bases de dados documentais para a realização de *"queries"*.

Seguindo a mesma linha de raciocínio, podemos dizer que a linguagem de selectores de que necessitamos para as restrições é também equivalente áquelas, podendo nós, desta maneira, tirar partido duma linguagem já existente com as seguintes vantagens: menos uma sintaxe no mundo das linguagens e consequentemente no nosso sistema, menos trabalho de implementação, menos trabalho de aprendisagem e de ensino, e por tudo isto menos confusão.

Depois de algum tempo de discussão (moderada pelo *W3C - World Wide Web Consortium*), começa a emergir algum consenso na utilização do XSLT [xslt], uma sublinguagem de padrões presente no XSL [xsl] - a proposta de normalização para a especificação de estilos a associar a documentos XML. O XSLT encontra-se já numa fase prévia ao lançamento de um standard e foi já alvo de um estudo formal por parte de Wadler [Wad99], apresentado na última conferência mundial da área (*"Markup Technologies 99"*), e onde ele define a linguagem usando semântica denotacional (formalismo de cariz funcional utilizado para especificar a sintaxe e a semântica de linguagens - [Paa95]).

Durante o trabalho desta tese, também se procedeu ao estudo de algumas destas linguagens (em particular todas as que já foram referidas) e, foi fácil constatar que o XSLT é um denominador comum de uma grande parte delas, aquelas que foram desenvolvidas a pensar em documentos estruturados, tratando-se portanto de uma linguagem específica. Houve, no entanto, uma linguagem que cativou a atenção do autor, pela sua simplicidade e recurso à teoria de conjuntos, a linguagem proposta por Tim Bray [Bray98] na *QL'98 - The Query Languages Workshop* designada por *Element Sets*. Um estudo mais atento da linguagem e do seu historial, revelou ser esta a especificação por detrás do conhecido motor de procura **Pat** comercializado pela *OpenText* e utilizado na maior parte dos primeiros portais da Internet.

Enquanto as linguagens do tipo XSLT assentam numa sintaxe concreta e específica, a *Element Sets* define uma notação abstracta baseada em cinco operadores da teoria de conjuntos: contido (*within*), contém (*including*), união (+), intersecção (^) e diferença (-). Bray argumenta ser capaz de especificar uma grande percentagem de queries que possam ser necessárias num sistema de arquivo documental à custa da combinação daqueles cinco operadores.

Numa primeira análise e a título comparativo, apresentam-se a seguir dois exemplos, uma query simples e uma mais complicada que irão ser especificadas respectivamente recorrendo a XSLT e a Element Sets.

Exemplo 9-6. Query simples

Pretende-se seleccionar todos os parágrafos (PARA) pertencentes à introdução (INTROD) que contenham uma ou mais notas de rodapé (FOOTNOTE) ou uma ou mais referências (REF) a outros elementos no documento.

Em *Element Sets* a query seria:

```
set1 = Set('PARA') within Set('INTROD')
set2 = set1 including Set('FOOTNOTE')
set3 = set1 including Set('REF')
set4 = set2 + set3
```

A função **Set** selecciona todos os elementos do tipo indicado no argumento. No fim, o resultado da query é dado pelo valor de **set4**.

Por outro lado, em *XSLT* teríamos:

```
INTROD/PARA[FOOTNOTE $or$ REF]
```

A query tem duas partes, a primeira de selecção de contexto, **INTROD/PARA**, que selecciona todos os parágrafos dentro da introdução, e a segunda parte de restrição desse contexto, **[FOOTNOTE or REF]**, que restringe o conjunto de elementos seleccionadas áqueles que tenham pelo menos um elemento do tipo **FOOTNOTE** ou **REF**.

Depois desta comparação, num caso muito simples, veja-se uma outra situação de interrogação mais elaborada.

Exemplo 9-7. Query mais complexa

Pretende-se agora, seleccionar todos os parágrafos da introdução que contenham uma referência ou uma nota de rodapé mas não ambos.

Em *Element Sets* a query seria:

```
set1 = Set('PARA') within Set('INTROD')
set2 = set1 including Set('FOOTNOTE')
set3 = set1 including Set('REF')
```

```
set4 = (set2 + set3) - (set2 ^ set3)
```

Apesar de complexa, foi fácil especificar esta query. Bastou excluir (diferença de conjuntos) os elementos resultantes da query anterior que continham ambos os elementos (intersecção de conjuntos), **REF** e **FOOTNOTE**.

Temos agora, a especificação em XSLT:

```
INTROD/PARA[(FOOTNOTE $and$ $not$ REF)  $or$ (REF $and$ $not$ FO-
OTNOTE)]
```

Do estudo comparativo realizado entre os dois tipos de linguagem, e do qual os dois exemplos acima fazem parte, podemos concluir que, em termos da operação de selecção, são mais ou menos equivalentes, não se tendo encontrado nenhuma situação que uma solucionasse e a outra não. Vão diferir é no método como fazem a selecção: o XSLT usa a árvore documental e toda a operação de selecção é feita em função dessa estrutura; a Element Sets, por outro lado, não usa a árvore documental, manipula o documento como um conjunto de elementos usando uma sintaxe mais universal. Mas esta diferença existe apenas perante o utilizador que usa a linguagem porque em termos de implementação não se pode fugir às travessias da árvore documental.

Tendo estudado os dois paradigmas era preciso tomar uma decisão: Qual a linguagem a adoptar/adaptar para a sub-linguagem de selecção no S4?

Ao contrário do que o leitor poderia supor nesta altura, a escolha não recaiu sobre a Element Sets mas sim sobre uma linguagem do tipo XSLT, a XQL - XML Query Language [RLS98]. Os motivos por detrás desta escolha são muito simples. Apesar dos paradigmas, em termos de selecção, serem equivalentes, as linguagens do tipo XSLT vão além da selecção, permitem ter um segundo nível de selecção baseado em restrições sobre o conteúdo. A ideia é utilizar a sintaxe desse segundo nível de selecção para especificar as restrições; *apenas será necessário alterar-lhe a semântica: em lugar dum conjunto de elementos, devolverá um valor booleano.*

Assim as nossas restrições serão compostas, como referido, por uma selecção e uma condição, mas integradas na mesma sintaxe. No resto desta secção, iremos descrever ambas as partes da linguagem.

9.2.2.1. Escolha da linguagem

A linguagem XSLT fornece um método bastante simples para descrever a classe de nodos que se quer seleccionar. É declarativa em lugar de procedimental. Apenas é preciso especificar o tipo dos nodos a procurar usando um tipo de padrões simples baseado na notação de directorias dum sistema de ficheiros (a sua estrutura é equivalente à de uma árvore documental). Por exemplo, **livro/autor**, significa: seleccionar todos os elementos do tipo autor contidos em elementos livro.

A XQL é uma extensão do XSLT. Adiciona operadores para a especificação de filtros, operações lógicas sobre conteúdo, indexação em conjuntos de elementos, e restrições sobre o conteúdo dos elementos. Basicamente, é uma notação para a especificação de operações de extracção de informação de documentos estruturados.

No nosso contexto, vamos dar-lhe uma aplicação diferente. Vamos usar uma parte para seleccionar elementos/contexto e outra para calcular condições de contexto sobre esses elementos. A condição de contexto vai ter um comportamento semelhante a um invariante: a linguagem de query selecciona todos os elementos para os quais se obtém um valor verdadeiro da condição; na nossa utilização da linguagem, uma mensagem de erro será emitida sempre que um dos elementos seleccionados faça com que se obtenha um valor falso da condição.

Como já foi dito, vamos começar por descrever operadores relacionados com a selecção mas a linha divisória entre selecção e restrição irá sendo diluída ao longo do texto, confundindo-se até, para os casos em que a integração das duas é muito forte.

9.2.2.1.1. Padrões e Contexto

Uma expressão de selecção é sempre avaliada em função dum contexto de procura. Um contexto de procura é um conjunto de nodos a que uma expressão se pode aplicar de modo a calcular o resultado. Todos os nodos no contexto de procura são filhos do mesmo nodo pai; o contexto de procura é constituído por todos os nodos que são filhos deste nodo pai e respectivos atributos mais os atributos do nodo pai.

As expressões de selecção poderão ser absolutas (o contexto é seleccionado em função do nodo raíz - "/"), ou relativas (o contexto é seleccionado em função do contexto actual - "."). Na especificação do contexto pode ainda ser usado o operador "//"com o significado de descendência recursiva.

Exemplo 9-8. Selecção de contextos

1. Seleccionar todos os elementos **autor** no contexto actual.
   ```
   ./autor
   ```
   Que é equivalente a:
   ```
   autor
   ```
2. Seleccionar o elemento raíz (**livro**) deste documento.
   ```
   /livro
   ```
3. Seleccionar todos os elementos **autor** em qualquer ponto do documento actual.
   ```
   //autor
   ```
4. Seleccionar todos os elementos **capítulo** cujo atributo **tema** é igual ao atributo **especialidade** de **livro**.
   ```
   capítulo[/livro/@especialidade = @tema]
   ```
5. Seleccionar todos os elementos **título** que estejam um ou mais níveis abaixo do contexto actual.
   ```
   .//título
   ```

9.2.2.1.2. Quantificador: todos

O operador "*"quando usado numa expressão de selecção selecciona todos os elementos nesse contexto.

Exemplo 9-9. Selecção com "*"

1. Seleccionar todos os elementos filhos de **autor**.
   ```
   autor/*
   ```
2. Seleccionar todos os elementos **nome** que sejam netos de **livro**.
   ```
   livro/*/nome
   ```

3. Seleccionar todos os elementos netos do contexto actual.
```
*/*
```
4. Seleccionar todos os elementos que tenham o atributo **identificador**.
```
*[@identificador]
```

9.2.2.1.3. Atributos

Como já se pôde observar nalguns exemplos, o nome de atributos é precedido por "@". Os atributos são tratados como sub-elementos, imparcialmente, sempre que possível. De notar que os atributos não podem ter sub-elementos pelo que não poderão ter operadores de contexto aplicados ao seu conteúdo (tal resultaria numa situação de erro sintáctico). Os atributos também não têm conceito de ordem, são por natureza anárquicos pelo que nenhum operador de indexação deverá ser-lhes aplicado.

Exemplo 9-10. Selecção com atributos

1. Seleccionar o atributo **valor** no contexto actual.
```
@valor
```
2. Seleccionar o atributo **dólar** de todos os elementos **preço** no contexto actual.
```
preço/@dólar
```
3. Seleccionar todos os elementos **capítulo** que tenham o atributo **língua**.
```
capítulo[@língua]
```
4. Seleccionar o atributo **língua** de todos os elementos **capítulo**.
```
capítulo/@língua
```
5. O próximo exemplo não é válido.
```
preço/@dólar/total
```

9.2.2.1.4. Filtro - sub-query

O resultado duma *query* pode ser refinado através de uma *sub-query* (restrição aplicada ao resultado da *query* principal), indicada entre "["e "]"(nos exemplos anteriores já apareceram várias sem nunca se ter explicado a sua sintaxe e semântica).

A *sub-query* é equivalente à cláusula SQL WHERE. O valor resultante da aplicação de uma *sub-query* é booleano e os elementos para os quais o valor final seja verdadeiro farão parte do resultado final.

Há operadores nas *sub-queries* que permitem testar o conteúdo de elementos e atributos. Vamos deixar a sua discussão para a secção seguinte pois iremos usá-los para especificar restrições.

Exemplo 9-11. Sub-query

1. Seleccionar todos os elementos **capítulo** que contenham pelo menos um elemento **excerto**.
   ```
   capítulo[excerto]
   ```
2. Seleccionar todos os elementos **título** pertencentes a elementos **capítulo** que tenham pelo menos um elemento excerto.
   ```
   capítulo[excerto]/titulo
   ```
3. Seleccionar todos os elementos **autor** pertencentes a elementos **artigo** que tenham pelo menos um elemento **excerto**, e onde autor tenha **email**.
   ```
   artigo[excerto]/autor[email]
   ```
4. Seleccionar todos os elementos **artigo** que contenham elementos **autor** com **email**.
   ```
   artigo[autor/email]
   ```
5. Seleccionar todos os elementos **artigo** que tenham um **autor** e um **título**.
   ```
   artigo[autor][titulo]
   ```

Como se pode observar nalguns destes exemplos, algumas das restrições que pretendemos colocar sobre os documentos podem ser especificadas com os

construtores e operadores já apresentados. A linha divisória entre a selecção e a restrição parece já um pouco diluída. Como os restantes operadores que se podem usar nas *sub-queries* já permitem interrogar conteúdos vamos discuti-los como uma aproximação à especificação de restrições.

9.2.2.1.5. Expressões booleanas

As expressões booleanas podem ser usadas nas sub-queries e estas, já nos permitem especificar condições contextuais como a restrição de valores a um domínio. Uma expressão booleana tem a seguinte forma:

```
val-esquerda $operador$ val-direita
```

Os operadores têm a forma **$operador$** e são normalmente binários: tomam como argumentos um valor à esquerda e um valor à direita.

Com estes operadores e o agrupamento por parentesis podem especificar-se queries bastante complexas, no nosso caso, restrições.

Exemplo 9-12. Operadores booleanos

1. Seleccionar todos os elementos **autor** que tenham um **email** e um **url**.
   ```
   autor[email $and$ url]
   ```
   No universo das queries, o resultado seria o conjunto de **autores** que tivessem **email** e **url**. Do nosso ponto de vista, e pensando na expressão como uma especificação duma restrição aplicada a todos os elementos **autor** o resultado seria uma mensagem de erro para todos os autores que não tivessem um **email** ou um **url**.

2. Seleccionar todos os elementos **autor** que tenham um **email** ou um **url** e pelo menos uma **publicação**.
   ```
   autor[(email $or$ url) $and$ publicação]
   ```
3. Seleccionar todos os elementos **autor** que tenham um **email** e nenhuma **publicação**.
   ```
   autor[email $and$ $not$ publicação]
   ```

4. Seleccionar todos os elementos **autor** que tenham pelo menos uma **publicação** e não tenham **email** nem **url**.
```
autor[publicação $and$ $not$ (email $or$ url)]
```

Como se pode ver, todas estas *queries* podem ser encaradas como restrições, apenas é preciso dar uma interpretação diferente aos resultados. A partir deste ponto, e uma vez que os operadores seguintes são dirigidos ao conteúdo, vamos, nos exemplos, deixar de nos referir a *queries*, e passar a fazê-lo em relação a restrições.

9.2.2.1.6. Equivalência

A igualdade é notada por **=** e a desigualdade por **!=**. Alternativamente podem ser usados **eq** e **neq**.

Podemos usar strings nas expressões desde que limitadas por aspas simples ou duplas.

Na comparação com o conteúdo de elementos o método **value()** está implícito. Por exemplo **nome='Carlos'** é uma abreviatura de **nome!value()='Carlos'**.

Como se disse no fim da última secção, vamos agora dar exemplos de restrições. Apesar de algumas parecerem um pouco "artificiais", elas são necessárias para exemplificar alguns dos operadores.

Exemplo 9-13. Equivalência

1. Garantir que todos os autores têm o sub-elemento **organização** preenchido com o valor **'U.Minho'**.
```
autor[organização = 'U.Minho']
```
2. Garantir que todos os elementos têm o atributo **língua** preenchido com o valor **'pt'**.
```
*[@língua = 'pt']
```
3. Imaginemos que temos vários documentos relativos aos curriculae de vários autores e que estes documentos foram gerados automaticamente a partir de uma

base de dados. Agora queremos garantir que todos as publicações têm o sub-elemento **nome** do elemento **autor** preenchido com um valor igual ao sub-elemento **nome** que figura no elemento **curriculum**.

```
publicação/autor[nome = /curriculum/nome]
```

Esta é uma restrição complexa uma vez que envolve a comparação de elementos localizados em diferentes ramos da árvore documental, daí a necessidade de recorrer a duas expressões de selecção.

4. Vamos agora para uma situação corrente numa instituição. O prazo para a entrega da proposta em que a equipe tem vindo a trabalhar foi ultrapassado. Em conversa com a entidade competente fomos informados de que os documentos ainda iriam a tempo se fossem entregues nos próximos dois dias mas as datas constantes nos documentos teriam de ser inferiores a '1999-12-22'.

   Podíamos adaptar o nosso editor para fazer essa verificação acrescentando-lhe a seguinte condição de contexto:

   ```
   data[. < '1999-12-22']
   ```

   O operador **.** selecciona o conteúdo do contexto actual. Assumiu-se que as datas estavam normalizadas, de qualquer modo, as que não estiverem não passarão pela restrição e serão detectadas.

A linguagem possui todos os operadores relacionais habituais, cuja utilização não foi aqui exemplificada, porém, a sua semântica é bem conhecida e o seu lugar na linguagem será bem especificado na próxima secção onde iremos definir formalmente a linguagem de restrições.

9.2.2.2. Definição da linguagem de restrições

As restrições serão especificadas junto com o DTD usando secções de comentário especiais (intocadas pelo parser SGML, podendo por isso ter uma sintaxe mais alargada). A restrição propriamente dita será especificada em CL (*"Constraint Language"*) que não é mais do que o subconjunto da XQL apresentada na secção anterior. A declaração duma restrição terá a seguinte forma:

```
<!- - CONSTRAINT  expressão-CL  acção - - >
```

Onde:

CONSTRAINT

É uma palavra reservada que serve para indicar ao interpretador das restrições que este não é um comentário SGML mas uma expressão em CL que deverá ser processada.

expressão-CL

É uma expressão em CL que especifica a condição de contexto a ser verificada.

acção

É a acção que deverá ser executada sempre que o cálculo da expressão resultar num valor falso. Nesta primeira versão, corresponderá apenas a uma string que conterá uma mensagem de erro a ser mostrada sempre que a acção fôr despoletada.

Tendo definido o enquadramento das restrições com o SGML, vai-se agora apresentar a definição formal da CL.

A CL não é um subconjunto da XQL, apenas utiliza uma parte da sintaxe da outra, semanticamente são completamente diferentes como se poderá ver à frente.

Iremos usar gramáticas de atributos para especificar a sua sintaxe e semântica. Para que a definição aqui apresentada não fique muito pesada será usada a sintaxe SSL apenas para a declaração dos atributos e especificação das suas equações de cálculo.

Nas funções semânticas que irão surgir, e que aparecerão devidamente especificadas (assinatura e descrição), é assumida uma estrutura global, o *grove* (Secção 6.2.4.1), que é construído durante o reconhecimento do documento. A maior parte destas funções são o que se denomina por vezes de *"tree-walkers"*; o *grove* é uma estrutura arbórea muitas vezes vista como uma lista generalizada e que é uma representação abstracta aceite para representar o documento anotado (conteúdo, anotação, atributos, ...). Cada restrição em CL, irá fazer cálculos sobre o *grove* e este é sempre assumido globalmente por cada restrição. Isto tem uma implicação: não há

endereçamentos relativos no *grove*, cada expressão de contexto deverá especificar um endereço absoluto.

Para uma melhor identificação as produções encontram-se numeradas de 1 a n, sendo 1 a produção correspondente ao axioma.

A estrutura seguida para a especificação de cada produção é a seguinte:

```
n <produção>
<declaração atributos>
<equações de cálculo>
```

e, por sua vez <**declaracao atributos**> é:

```
<Símbolo> {(modo tipo nome-atributo;)+};
```

Para efeitos de legibilidade, os atributos são declarados na produção correspondente à sua primeira utilização.

Assim, a nossa gramática de atributos é:

```
1. CLexp → Contexto Cond

CLexp
  { syn STR contexto;
    syn STR condicao;
    syn MAP[IDENT,BOOL] resultado; };
Contexto
  { syn STR fc; }; /* final context exp */
Cond
  { syn STR fb; }; /* final boolean exp */

  { $$.contexto = Contexto.fc;
    $$.condicao = Cond.fb;
    $$.resultado = Valida($$.contexto, $$.condicao); };
```

Uma expressão em CL é estruturalmente constituída por duas partes: um selector de contexto, e uma condição. Portanto, o símbolo não-terminal (NT) **CLexp** irá ter dois atributos sintetizados; um que sintetizará a expressão de contexto e outro que sintetizará a condição: **contexto**, **condicao**. O cálculo da expressão de contexto deverá resultar num conjunto de nodos do *grove* (muitas vezes singular). Por outro lado, o cálculo da expressão condicional, quando

aplicada aos nodos anteriores resultará num conjunto de valores booleanos. Ambas as expressões são sintetizadas num formato pré-fixo (à semelhança de uma linguagem funcional tipo LIST).

Quando da expressão de contexto resultar um conjunto de nodos, a condição será avaliada para cada um deles individualmente.

A função semântica valida recebe uma expressão de contexto que irá seleccionar um conjunto de nodos no *grove*, recebe ainda uma condição e vai calcular para cada nodo o valor booleano da condição. No fim, produz um resultado que é uma correspondência entre o identificador do nodo e o resultado booleano do cálculo da condição sobre esse nodo.

```
2. Contexto → '/' RelContexto

RelContexto { syn STR fc;
              inh STR ic; };
  { $$.fc = RelContexto.fc;
    RelContexto.ic = "ROOT"; };
```

O NT RelContexto tem um atributo herdado (ic), necessário em todos os símbolos com produções recursivas com dependências semânticas e que tem o valor sintetizado na iteração anterior.

A expressão de contexto **'/'** representa o nodo raíz do *grove*. Isso é indicado através da constante "**ROOT**".

```
3. Contexto → RelContexto
  { $$.fc = RelContexto.fc;
    RelContexto.ic = "ROOT"; };
```

Quando a expressão não começa por **'/'** assume-se que o contexto inicial é "**ROOT**".

```
4. RelContexto → elem-id '/' RelContexto

elem-id
  { syn IDENT v; };

  { $$.fc = RelContexto$2.fc;
    RelContexto$2.ic = "(Sel " + elem-id.v + $$.ic + ")"; };
```

elem-id é um símbolo terminal que tem um atributo intrínseco (proveniente da análise léxica) cujo valor corresponde ao nome dum elemento.

A função **Sel** tem a seguinte assinatura:

```
Sel : nome identificador-set → identificador-set
```

Ou seja, funciona como um filtro. Restringe o conjunto de nodos passado como segundo parâmetro, aos nodos que tenham o nome igual ao primeiro parâmetro.

Note-se a diferença entre nome e identificador. Pode haver vários nodos com o mesmo nome, por exemplo *capítulo*, mas cada um desses nodos tem um identificador único que é calculado pelo sistema no momento da construção do *grove*.

```
5. RelContexto → elem-id '//' '//' RelContexto
  { $$.fc = RelContexto$2.fc;
    RelContexto$2.ic = "(Descendentes " + elem-
   id.v + $$.ic + ")"; };
```

A função **Descendentes** tem a seguinte assinatura:

```
Descendentes : identificador-set → identificador-set
```

Recebe um conjunto de identificadores como entrada e dá como resultado o conjunto de identificadores de todos os nodos dos *sub-groves* com raíz correspondente aos nodos referenciados pelos identificadores entrados.

```
6. RelContexto → '*'
  { $$.fc = "(Filhos " + $$.ic + ")"; };
```

A função **Filhos** tem a seguinte assinatura:

```
Filhos : identificador-set → identificador-set
```

Recebe um conjunto de identificadores (normalmente singular) e devolve um conjunto de identificadores correspondentes aos nodos filhos dos nodos correspondentes aos identificadores de entrada.

```
7. RelContexto → elem-id
  { $$.fc = "(Sel " + elem-id.v + " " + $$.ic + ")"; };
8. RelContexto → '@' Atrib

Atrib
```

```
  { syn IDENT v; };

  { $$.fc = "(Atributo " + Atrib.v + " " + $$.ic + ")"; };
```

A função **Atributo** tem a seguinte assinatura:

```
    Atributo : atrib-identificador identificador-
set → atrib-identificador-set
```

Esta função selecciona identificadores correspondentes a **Atrib.v** (nome de um atributo) que sejam filhos de algum dos nodos em **identificador-set**.

```
9. Atrib → atrib-id

atrib-id
  { syn IDENT v; };

  { $$.v = atrib-id.v; };
```

atrib-id.v é um atributo intrínseco proveniente da análise léxica.

```
10. Atrib → '*'
  { $$.v = "ALL"; };
```

ALL é um identificador especial, selecciona todos os atributos. A sua utilidade no contexto da linguagem de restrições também será discutida numa futura optimização.

```
11. Cond → '[' Exp ']'
  { $$.fb = Exp.fb; };
```

Primeira produção respeitante à derivação de condição.

Como já foi dito, iremos sintetizar uma expressão na forma pré-fixa com parentesis para a condição.

```
12. Exp → Termo OR Exp

Termo
  { syn STR fb; };

  { $$.fb = "(Or "+Termo.fb+" "+Exp.fb+")"; };
```

As funções presentes na expressão que está a ser sintetizada são as operações lógicas e aritméticas normais pelo que não iremos aqui defini-las.

```
13. Exp → Termo
  { $$.fb = Termo.fb; };
14. Termo → Factor AND Termo

Factor
  { syn STR fb; };

  { $$.fb = "(And "+Factor.fb+" "+Termo.fb+")"; };
15. Termo → Factor
  { $$.fb = Factor.fb; };
16. Factor → SubExp

SubExp
  { syn STR fb; };

  { $$.fb = SubExp.fb; };
17. Factor → NOT SubExp
  { $$.fb = "(Not "+SubExp.fb+")"; };
18. Factor → '(' Exp ')'
  { $$.fb = Exp.fb; };
19. SubExp → Operando comp-op Operando

Operando
  { syn STR valor; };
comp-op
  {syn STR op; };

  { $$.fb = "("+comp-op.op+ " "+
            Operando.valor+" "+" "+Operando.valor+")"; };
```

com-op.op é um atributo intrínseco sintetizado da análise léxica e que poderá ter um dos valores: =, !=, <, >, <=, >=.

```
20. Operando → elem-id
  { $$.valor = "(Conteúdo "+elem-id.v+")"; };
```

A função **Conteúdo** tem a seguinte assinatura:

```
Conteúdo : identificador-set → STR-seq
```

Um dos componentes dos nodos tipo elemento é o seu conteúdo (relembrar a estrutura dum grove). Esta função extrai esse conteúdo dos nodos correspondentes aos identificadores e devolve-os numa lista.

```
21. Operando → '.'

Cond, Exp, Termo, Factor, SubExp, Operando
  {inh STR ic; };

  { $$.valor = "(Conteúdo(eval("+Operando.ic+"))"; };
```

Operando tem um atributo herdado de nome **ic** que tem a expressão de contexto sintetizada no axioma. Este valor é passado ao longo da árvore através deste atributo herdado que todos aqueles NT acima indicados têm. Nas produções anteriores, não se colocaram as equações de cópia (*"copy rules"*) para não tornar a gramática desnecessariamente pesada.

A função **eval** recebe uma expressão de contexto e dá como resultado um conjunto de identificadores.

Tem a seguinte assinatura:

```
eval : contexto-exp → identificador-set
```

```
22. Operando → '@' atrib-id
  { $$.valor = "(Valor "+atrib-id.v+")"; };
```

A função **Valor** é semelhante à função **Conteúdo**, só que aplicada a atributos. **Valor** devolve uma lista correspondente aos valores dos atributos cujos identificadores lhe são passados como parâmetro.

Tem a seguinte assinatura:

```
Valor : atrib-identificador-set → STR
```

```
23. Operando → Expnum

Expnum
  {syn STR vnum; };
```

```
  { $$.valor = Expnum.vnum; };
```

24. `Operando → string`

```
  { $$.valor = string.v; };
```

string.v é um atributo intrínseco proveniente da análise léxica.

25. `Operando → Selec`

```
Selec
  { syn STR fc; };

  { $$.valor =" (Conteúdo(eval "+Selec.fc+"))"; };
```

O NT **Selec** representa o *path* para um elemento ou atributo noutro ponto da árvore que não o nodo corrente ou algum da sua descendência. Será usado para derivar condições que comparam valores localizados em *sub-groves* distintos.

26. `Operando → '(' SubExp ')'`

```
  { $$.valor = SubExp.fb; };
```

27. `Selec → '//' RelSelec`

```
RelSelec
  { syn STR fc;
    inh STR ic; };

  { $$.fc = RelSelec.fc;
    RelSelec.ic = "ROOT"; };
```

A construção da expressão de contexto para **Selec** será muito semelhante à que foi usada para o NT **Contexto**.

28. `Selec → RelSelec`

```
  { $$.fc = RelSelec.fc;
    RelSelec.ic = ; };
```

29. `RelSelc → elem-id '//' RelSelec`

```
  { $$.fc = RelContexto$2.fc;
    RelContexto$2.ic = "(Sel " + elem-id.v + $$.ic + ")"; };
```

30. `RelSelc → elem-id`

```
  { $$.fc = "(Sel " + elem-id.v + " " + $$.ic + ")"; };
```

31. `RelSelc → '@' atrib-id`

```
  { $$.fc = "(Atributo " + Atrib.v + " " + $$.ic + ")"; };
32. Expnum → TermoN OpAd ExpNum

TermoN { syn STR vnum; };

  { $$.vnum = (OpAd.op == '+') ?
                "(+ "+TermoN.vnum+" "+Expnum$2.vnum+")" :
                "(- "+TermoN.vnum+" "+Expnum$2.vnum+")"; };
```

As expressões aritméticas são semelhantes às expressões lógicas. O esquema de prioridades reflectido na gramática é o mesmo, apenas muda a simbologia dos operadores.

```
33. Expnum → TermoN
  { $$.vnum = TermoN.vnum; };
34. TermoN → FactorN OpMul TermoN

FactorN { syn STR vnum; };

  { $$.vnum = (OpMul.op == '*') ?
                "(* "+FactorN.vnum+" "+TermoN$2.vnum+")" :
                "(/ "+FactorN.vnum+" "+TermoN$2.vnum+")"; };
35. TermoN → FactorN
  { $$.vnum = FactorN.vnum; };
36. FactorN → num
  { $$.vnum = num.v; };
37. FactorN → '(' Expnum ')'
  { $$.vnum = Expnum.vnum; };
```

Fica assim definida a primeira versão da linguagem de restrições. Haverá com certeza optimizações a fazer. Um dos aspectos a ter em conta no trabalho futuro será analisar a utilidade dos operadores definidos, alguns deles introduzem muita complexidade a nível semântico, caso venha a concluir-se que não são muito necessários serão os primeiros a sair.

Na secção seguinte, vamos ilustrar a utilização da linguagem, aplicando-a a exemplos que têm vindo a ser introduzidos.

9.2.2.3. Aplicação aos casos de estudo apresentados

Ao longo dos capítulos anteriores, foram apresentados vários problemas onde era necessário especificar condições de contexto que teriam de ser verificadas de modo a limitar os erros no conteúdo dos respectivos documentos. Vamos, agora, ver como se especifica cada uma daquelas restrições com a linguagem que acabamos de definir.

Iremos identificar cada exemplo e indicar a solução proposta na altura em que foi apresentado:

Reis e Decretos

Este foi o exemplo utilizado no capítulo anterior para ilustrar a abordagem à especificação de restrições com modelos abstractos. Na altura, especificaram-se as restrições em CAMILA da seguinte forma:

```
rei(r) =
{ if(nome_(r) notin PessFamDB -
> nome_(r) ++ "Não existe..."),
  if(datan_(r) > datam_(r) -
> nome_(r) ++ "Morreu antes de nascer."),
  if(datam_(r) - datan(r) > 120 -
> nome_(r) ++ "Viveu demais!"),
  if(!all( x<-decreto_l(r) : datan_(r) < data_(x) /\ da-
ta_(x) < datam_(r))
          -
> nome_(r) ++ "fez um decreto fora da sua vida!")
};
```

Usando CL especificaríamos as mesmas restrições assim:

```
<!- - CONSTRAINT  /rei[datan/@valor < datam/@valor]
                 "Morreu antes de nascer" - ->

<!- - CONSTRAINT  /rei[datan/@valor-datam/@valor < 120]
                 "Viveu muito tempo" - ->

<!- -
CONSTRAINT  /rei/decreto[(data/@valor < rei/datam/@valor)
                                $or$ (da-
ta/@valor > rei/datan/@valor)]
```

```
                "Fez decretos fora da sua vida" - ->
```

Não foi possível exprimir a primeira condição pois nesta versão da CL não foi prevista a ligação a bases de dados.

Na comparação das datas usou-se o atributo valor pois é aqui que está o valor normalizado da data (lembrar discussão sobre a normalização de conteúdos: Exemplo 7-2).

Arqueologia

No Capítulo 7, pegamos num caso real em que estávamos a trabalhar, a edição dos "Arqueosítios do Noroeste português", para exemplificar a necessidade de invariantes durante a edição de documentos. Especificamente, tínhamos dois campos, latitude e longitude, cujo conteúdo deveria estar compreendido dentro de determinado domínio. Estava prevista uma ligação a um sistema de informação geográfico, e aquele domínio seria definido pelas coordenadas das extremidades do mapa.

Em CL, seria simples especificar esta restrição:

```
<!- - CONSTRAINT  latitude[(.>39) $and$ (.<42)]
                "Latitude errada!" - ->

<!- - CONSTRAINT  longitude[(.>0) $and$ (.<55)]
                "Longitude errada!" - ->
```

Ou ainda, e relembrando que um contexto de procura é constituído por todos os nodos que são filhos do nodo pai indicado, e respectivos atributos, mais os atributos do nodo pai:

```
<!- -
CONSTRAINT  arqueositio[(latitude>39) $and$ (latitude<42)]
                "Latitude errada!" - ->

<!- - CONSTRAINT  arqueosi-
tio[(longitude>0) $and$ (longitude<55)]
                "Longitude errada!" - ->
```

Esta última solução só testa acondição para os elementos latitude e longitude do arqueosítio. Se outros houver ela não lhes será aplicada.

9.2.2.4. Estado actual do S4

Neste momento, o estado de implementação do S4 é o seguinte:

- o editor de DTDs está implementado sem restrições. No entanto, no decorrer desta tese as regras de conversão de DTDs para GAs foram revistas várias vezes daí resultando a versão em apêndice (Apêndice C), que é muito diferente da que se encontra implementada [Lop98]; logo uma reimplementação será necessária.
- a linguagem de restrições tem agora a sua sintaxe e semântica especificadas numa gramática de atributos (a sua implementação não será difícil).
- o editor de estilos está implementado [Sou98], mas a ligação ao editor de DTDs necessita de ser trabalhada; a linguagem para a especificação de estilos deverá também ser revista o que reconduzirá a nova implementação.

Neste capítulo, apresentamos o contributo mais importante da tese: a definição formal da linguagem de restrições e a construção duma arquitectura funcional que integra o processamento desta nova linguagem com o ciclo de vida dos documentos SGML.

A tese termina no próximo capítulo com uma síntese do trabalho realizado e deixando alguns indicadores de trabalho futuro.

Capítulo 10. Conclusão

Neste documento descreve-se o trabalho realizado pelo autor na sua tese de doutoramento.

O trabalho teve duas grandes linhas orientadoras. A estruturação de documentos, como a maneira de os tornar mais "ricos"e mais "vivos". E, a especificação da semântica dos documentos, desde a aparência visual até à interpretação (significado) do seu conteúdo como um meio de melhorar a qualidade na produção documental electrónica. No fim, estas duas linhas acabaram por convergir na elaboração dum novo modelo de processamento documental.

As vantagens dos documentos estruturados foram apresentadas e os passos para a implementação de um sistema de produção de documentos estruturados foram descritos.

Depois, apresentou-se o conjunto de necessidades e requisitos actuais que se podem colocar a um sistema destes e analisou-se aquilo que se designou por "semântica dos documentos". As necessidades identificadas estão relacionadas com o problema da qualidade de conteúdos na publicação electrónica. A qualidade em publicações electrónicas pode ser analisada segundo vários parâmetros, desde o aspecto visual , o linguístico e literário, à correcção da informação (significado, semântica). A tecnologia existente permite de alguma forma automatizar e normalizar todos estes aspectos, excepto o último. Foi no desenvolvimento de uma solução para este problema que se centrou esta dissertação: como adicionar semântica estática (condições contextuais ou invariantes) aos documentos; e como processar esta semântica estática dum modo integrado com a tecnologia existente.

Foram apresentadas duas vias para a solução da especificação e processamento da semântica estática, a primeira segue uma aproximação via modelos abstractos, a outra, uma aproximação via gramáticas de atributos.

As duas abordagens surgiram em alturas diferentes e seguiram duas direcções diferentes. Enquanto na abordagem com modelos abstractos optamos por definir modelos específicos (deriva-se o modelo do DTD), na abordagem com gramáticas de atributos construímos a representação abstracta habitual para um documento, o *grove*, e assumimos a existência duma máquina virtual que trabalha sobre o *grove* com quatro funções de travessia.

A primeira abordagem revelou-se mais fraca porque:

- o analista tem que especificar as restrições em CAMILA (tem que conhecer mais uma sintaxe)
- a especificação de restrições está dependente do modelo (este é específico)
- as travessias da estrutura (uma vez que esta é específica) têm que ser especificadas em CAMILA pelo analista

Todas estas razões não vão contra a metodologia utilizada (especificação via modelos abstractos) mas sim contra a utilização dela (modelo específico, processamento específico).

A segunda abordagem resultou muito melhor porque:

- a informação é sempre carregada na mesma estrutura abstracta, o *grove*
- a linguagem definida pelo autor e na qual são especificadas as restrições tem uma sintaxe orientada à manipulação do grove, bastante simples e acessível
- o analista não precisa de programar nenhuma travessia ou outro tipo de processamento sobre o grove; o processador da linguagem de restrições traduz estas para instruções duma máquina virtual que tem nela implementadas todas as travessias e processamentos necessários

Da implementação desta segunda abordagem resultou o sistema S4, um sistema de processamento documental que integra e implementa as ideias defendidas: especificação de sintaxe, de estilo e de semântica. Como já foi referido (Capítulo 9), o S4 está ainda num estado embrionário de implementação: existe um protótipo do editor de DTDs; existe um protótipo do editor de estilos; a linguagem de especificação para as restrições foi definida nesta dissertação. Nos próximos tempos, vai-se lançar um novo projecto de implementação do S4, onde se irão combinar os dois protótipos existentes e adicionar o terceiro componente para tratamento das restrições (editor + processador).

Neste momento, está em desenvolvimento uma terceira abordagem resultante da experiência prática que o autor tem vindo a adquirir com o desenvolvimento de projectos reais nesta área. Esta abordagem passa pela utilização de um motor de transformação "Open Source"; as restrições são especificadas na linguagem definida pelo utilizador e o motor é alterado para que durante a travessia do documento teste se há restrições a ser calculadas; se houver, processa-as e faz depender o resto da execução do resultado desse processamento.

10.1. Trabalho Futuro

Depois de ler esta dissertação (apêndices incluídos), é fácil concluir que esta é uma área em permanente ebulição e que novas linguagens e metodologias surgem a cada dia que passa. Porém, há duas etapas do ciclo de vida dos documentos que não têm tido a atenção das outras, são elas: a análise e o armazenamento.

Relativamente à análise, as metodologias vão aparecendo [MA96]. O mesmo não acontece com ferramentas que as implementem (ferramentas visuais para especificação de estrutura, motores de inferência gramatical): são quase inexistentes! Esta é portanto uma linha de investigação em aberto que o autor pretende explorar no futuro e se possível integrar resultados daí resultantes no S4.

Em relação ao armazenamento, apesar da grande variedade de produtos existente [DuC98], a verdade é que nenhuma das equipes de desenvolvimento por detrás deles expôs como é que o seu sistema armazena os documentos, que mecanismos implementou para tirar partido da sua estrutura, etc. Isto leva-nos a crer que as suas soluções poderão ser soluções de implementação e não metodologias que possam ser aplicadas genericamente. Há aqui, portanto, trabalho a desenvolver, que tipo de base de dados utilizar, como criar os índices de modo a tirar partido da estrutura dos documentos, que mecanismos de endereçamento utilizar, etc. É claro, que o S4 também irá, no futuro, necessitar dum sistema de armazenamento, pelo que este trabalho de investigação deverá também ser integrado no projecto global do S4.

O S4, por sua vez, será integrado com outras linhas de investigação relacionadas, tais como: criação e manutenção de catálogos para documentos estruturados; thesaurus; e encicolpédias electrónicas. Este novo projecto de investigação, do grupo de investigação onde está integrado o autor da presente dissertação, será designado por *Scriptorium*.

Apêndice A. Como foi produzida esta tese

Porquê este apêndice? Não é esta tese igual a tantas outras?

A resposta poderia ser sim, noutro local ou noutra fatia temporal, mas neste caso é não. Neste momento, é provavelmente a primeira tese a ser produzida de raíz na língua portuguesa usando a tecnologia SGML. Como tal, é importante que fiquem registados os vários passos dados na realização deste objectivo.

A publicação desta tese constituiu em si um pequeno projecto cujas etapas se enumeram a seguir:

1. Análise da estrutura da dissertação e definição do DTD.
2. Escolha do editor estruturado de SGML.
3. Escolha de um sistema de formatação: stylesheets, processadores, conversores, ...
4. Escolha do formato de saída, HTML, RTF, TeX/PDF, a ser produzido para visualização e impressão do documento.
5. Definição de uma estratégia para o desenvolvimento (escrita e impressão) modular da tese.

Vamos agora ver com mais detalhe cada uma das etapas.

A.1. O DTD

O primeiro passo dum projecto deste tipo consiste sempre na análise do tipo de documento que se quer desenvolver para se decidir pela escolha dum DTD já existente para essa classe, ou pela criação dum novo. Uma vez que criar um DTD pode vir a ser uma tarefa árdua, é sempre aconselhável procurar um DTD já desenvolvido que sirva as nossas necessidades (até porque juntamente com o DTD é, normalmente, desenvolvido um conjunto de ferramentas que nos podem fazer ganhar tempo noutras etapas do projecto).

Em 1998, David Megginson publicou uma análise que fez aos DTDs disponíveis no mercado [Meg98]. Nessa análise, ele catalogou cada DTD seguindo alguns parâmetros, nomeadamente: facilidade de aprendizagem; facilidade de utilização; e facilidade no processamento. A facilidade na aprendizagem é medida em função de alguns parâmetros como o tamanho (número de elementos e atributos), consistência e o ser ou não intuitivo. A facilidade na utilização está quase exclusivamente relacionada com a organização em níveis do DTD (corresponde ao esforço físico de criação dum contexto, instanciação de atributos, ...). A facilidade no processamento está muito relacionada com os dois items anteriores: diversidade de contextos, previsão, análise do DTD.

A análise de mercado de Megginson reduziu o domínio de escolha a cinco possibilidades, que correspondem às maiores percentagens de utilização por parte da indústria de publicação electrónica:

ISO 12083

> Este devia ter sido o DTD da indústria da publicação electrónica. Foi desenvolvido por um grupo de especialistas da publicação electrónica para a preparação e anotação de manuscritos electrónicos. Resultou num DTD de dimensão considerável e com grande complexidade o que aliado a algumas incapacidades como a falta de consenso na anotação da matemática fez com que não tivesse grande aceitação.

DocBook

> DTD utilizado e desenvolvido inicialmente pela IBM. Hoje é um dos mais difundidos, com direito até a publicações sobre a sua concepção e parametrização [WM99]. Entre muitos outros é utilizado pela IBM, pela O'Reilly, e na maior parte das publicações SGML científicas.
>
> Foi um dos DTDs a ir para a balança, na implementação desta tese.

Text Encoding Initiative (TEI)

> Este DTD resulta de um esforço de cinco anos por parte de membros da comunidade académica e de investigação. Iniciado em 1987, o projecto desenvolveu-se à custa da vontade de pessoas ligadas às áreas de humanísticas, ciências sociais e letras, que utilizavam computadores. Este grupo percebeu as

vantagens de ter um mesmo conjunto de anotações para a sua produção documental: reduzir a diversidade de conjuntos de anotações (DTDs) existentes e em uso, simplificar o processamento feito pelo computador, e encorajar o intercâmbio de textos electrónicos.

O DTD foi sendo desenvolvido modularmente (segundo os autores seguindo o modelo da "Chicago Pizza") e com o decorrer dos anos foi engrossando a sua estrutura aumentando desta maneira o número de áreas de aplicação [SB94]. Hoje , dos DTDs mais utilizados é o maior e é a referência a utilizar pela comunidade de humanísticas, artes, letras e ciências sociais.

Também foi para a balança na implementação desta tese.

MIL-STD-38784 (CALS)

Um dos primeiros grupos a adoptar o SGML como formato de base para a produção documental foi o exército americano. Exército e organizações governamentais investiram meios no desenvolvimento de DTDs para documentos como os manuais de manutenção de aeronaves e outro tipo de equipamento pesado. Desse esforço resultaram alguns bons DTDs. O que está aqui em questão é para a produção de manuscritos mas a falta de informação sobre a sua utilização é muito grande no domínio público.

No entanto, houve um mini-DTD desenvolvido por este grupo que se tornou um standard: designado por **CALS-TABLE-MODEL** é um pequeno DTD incluído na maior parte dos outros para tratar a informação com estrutura tabular.

HTML 4.0

Como a produção documental para a Internet não pára de aumentar e, apesar de todos os avisos, a maior parte dessa documentação continua a ser produzida em HTML, este ainda é um dos DTDs com maior número de utilizadores [html4.0]. Por razões óbvias levantadas ao longo desta tese (mistura de conteúdo e forma, conjunto de anotações limitado, ...), este DTD não foi tido em conta na nossa selecção.

Depois desta pequena exposição, não deverá ser difícil concluir que a escolha estaria entre o Docbook e o TEI. Foi uma decisão trabalhosa. Testaram-se ambos

em vários documentos e com várias ferramentas. Destes testes resultaram as seguintes conclusões:

- Ambos têm um tamanho considerável, o que tem algumas implicações na facilidade de utilização.
- Há um maior suporte para o Docbook. Quase todas as ferramentas de SGML/XML trazem uma versão desse DTD, pronta a usar. Existe algum suporte para o TEI, mas a sua utilização nunca foi tão directa como a do outro.
- Para o Docbook existem uma série de ferramentas de processamento mas, mais importante, existe um projecto a decorrer de desenvolvimento e aperfeiçoamento dum conjunto de stylesheets DSSSL.

  No caso do TEI, e reportando-nos à época em que se tomou esta decisão (1996/97), existiam apenas algumas scripts em Perl para o processar, e o projecto de criação de stylesheets DSSSL estava ainda numa fase de estudo de viabilidade.

  Dado que, a escolha teria que ter em atenção o que viria à frente, foi principalmente o último ponto (a existência de stylesheets DSSSL) que fez com que a escolha recaísse sobre o Docbook.

A.2. O Editor

Depois de ter escolhido o DTD, era necessário decidir sobre qual o ambiente de edição a utilizar. Um editor normal não serviria pois não permitiria tirar partido da utilização do SGML. Assim, foi necessário procurar um editor SGML na oferta do mercado de então (hoje as possibilidades são mais variadas).

Para além dos parâmetros normais a ter em conta na selecção dum editor de texto (velocidade, suporte gráfico, manuseamento de rato), há outros que é necessário ter em conta quando o editor vai ser usado para editar SGML:

- O editor deve ser capaz de reconhecer no texto as anotações que marcam o início e fim dos elementos, os atributos, e as entidades, definidos por um DTD.
- O editor deve fornecer ajuda contextual ao utilizador. Preveni-lo sempre que este incorrer em erro, fornecer a lista de anotações válidas num determinado ponto do documento, e localizá-lo na árvore estrutural do documento.

- O editor deve permitir que o utilizador se desloque, ao longo do texto, de anotação em anotação.

A seguir enumeram-se os editores analisados. Nalguns casos foram mesmo testados em situações reais, como a escrita de artigos que seriam enviados para conferências SGML (nessas conferências, os artigos só são aceites para avaliação se forem redigidos de acordo com um determinado DTD, o que obriga à utilização dum editor SGML e/ou a uma validação posterior com um parser).

Emacs + PSGML

Na altura, esta plataforma representava a única solução sem custos de aquisição e consistia no editor Emacs (disponível para quase todas as plataformas) parametrizado com o modo PSGML [Sta99] (o Emacs é um editor aberto que permite que a maior parte das suas funcionalidades sejam reescritas; o PSGML é uma configuração para adaptar o editor ao SGML: introduz a ajuda contextual na edição e acrescenta uma série de menus para gerir as anotações).

Os primeiros artigos submetidos a conferências de SGML foram escritos usando esta plataforma. Os resultados eram satisfatórios. Começaram a surgir problemas com as últimas versões do Emacs, o modo PSGML não se instalava correctamente e o funcionamento do editor ficava um pouco baralhado. Houve que procurar outra plataforma.

FrameMaker+SGML

Depois de contactada, a Adobe enviou uma cópia do produto que se podia utilizar em pequenos projectos para efeitos de avaliação.

O FrameMaker era, nas versões anteriores, um editor/processador de texto muito poderoso, esta versão com suporte para SGML manteve essas características.

Testámo-lo num projecto de estágio dum aluno finalista que redigiu o respectivo relatório em SGML usando a ferramenta. Pudemos concluir o seguinte: O ambiente de trabalho é muito bom, ajuda contextual gráfica, o utilizador pode seleccionar o modo como quer editar o texto (como um editor normal ou trabalhando sobre a árvore documental), é WYSIWYG permitindo mesmo ter imagens (uma limitação de muitos editores SGML), fecha o ciclo

de produção permitindo a partir duma especificação de estilo (numa linguagem própria) que o utilizador gere o resultado final que entender (PDF,PS, ...).

Tem, no entanto, uma grande desvantagem, a configuração do editor é difícil e a linguagem para a especificação de estilo também não é acessível. Como estes motivos criavam atrasos, esta solução foi colocada de lado.

WordPerfect+SGML

Esta plataforma foi-nos cedida pela Corel numa conferência SGML. O objectivo deles foi arranjar uma equipe de testes sem custos (o produto cedo revelou que ainda precisava de uns acabamentos).

Podemos dizer que se trata de um editor já conhecido, com umas pequenas adaptações para trabalhar com o SGML. Assim, perde um pouco quando comparado com ferramentas semelhantes desenvolvidas de raíz a pensar no SGML. De notar, que esta plataforma serve apenas para edição, apesar de ser WYSIWYG os resultados impressos não são satisfatórios.

A versão testada, WordPerfect 8.0, tinha alguns problemas com as figuras e alguns ambientes onde a preservação dos caracteres brancos era essencial. Na altura, uma nova versão estava em desenvolvimento, na qual, o autor fez parte da equipe de testes tendo apontado alguns problemas detectados na utilização da versão anterior.

Depois de algumas utilizações e devido a alguns problemas (pequenos mas que por vezes assumiam outras dimensões), acabamos por colocar esta solução de lado.

Adept Editor

Este produto da Arbortext, talvez seja o melhor editor do mercado. Não foi possível testá-lo "em casa"uma vez que a empresa não foi receptiva a um período de experimentação ou mesmo a uma utilização no ensino e uma licença do produto custa dez vezes mais que a licença dum editor médio.

AuthorEditor

Simples, eficaz, barato. Foi a nossa escolha. O produto foi descontinuado depois da Softquad ter vendido os seus direitos, no entanto, a mesma empresa

lançou um novo produto compatível com todos os documentos SGML criados pelo outro, mas mais vocacionado para documentos XML: o **XMetal**.

O XMetal também está a ser utilizado nos nossos projectos. Não está a ser utilizado para este da tese porque apareceu numa fase em que a escrita desta ia avançada e, apesar do conteúdo não estar dependente da plataforma (lembrar uma das características fundamentais do SGML), a mudança exigiria uma adapatação da metodologia de trabalho que não se justificava.

Ao longo desta exposição, discutiram-se os editores SGML com mais utilização e com mais presença no mercado. A nossa escolha foi motivada em primeiro pelo custo e em segundo pela simplicidade e suporte da filosofia SGML.

A.3. O Formatador

Além da edição dirigida pela sintaxe SGML, a validação estrutural dum documento é assegurada por um parser (SP) associado ao Editor. Assim, à saída do editor temos um documento SGML puro (texto com anotações e sintaticamente válido). Há que processá-lo para se obter o documento num formato ideal para publicação. O ideal é que este processamento seja o mais standard possível, sem haver necessidade de muitas parametrizações ou mesmo programações específicas.

Pretendia-se que esta tese fosse publicada em papel (é o normal e obrigatório), em CDROM e na Internet. O que implicava a geração do documento em formato: para papel, RTF, PDF ou PS; para a Internet e CDROM, HTML. Tínhamos, portanto à partida dois formatos bem diferentes, o normal para papel em que um documento é estruturado em páginas, e o outro, em HTML, não tem conceito de página física (não há dimensões limites), é hipertexto (consegue-se navegar no documento seguindo as referências e entradas de índices). Depois do que foi apresentado na tese, havia também a convicção de se usar DSSSL para especificar o estilo no qual o documento deveria ser convertido.

Uma das razões que fez a escolha recair sobre o DTD Docbook foi a existência dum projecto de desenvolvimento dum conjunto de stylesheets em DSSSL para processar documentos escritos segundo aquele DTD. Este projecto de desenvolvimento está a ser conduzido por Norman Walsh com a designação *"Modular Docbook DSSSL Stylesheets"* [Wal2000] e tem recebido contribuições de várias dezenas de

utilizadores o que faz com que novas e melhoradas versões sejam lançadas cada quinze dias (quando aderimos ao esquema começamos por utilizar a versão 1.05, agora já estamos a utilizar a versão 1.50). A compatibilidade entre versões é garantida desde que algum cuidado seja tido em conta aquando da sua utilização. De qualquer modo, a linguagem de especificação não é alterada, o que acontece é que elementos que não eram suportados pela especificação têm vindo a ser adicionados.

Como o Docbook nunca tinha sido usado para formatar um documento escrito em português foi preciso adicionar o suporte da língua portuguesa, tarefa que logo me dispuz a fazer e que uma vez terminada, foi imediatamente adicionada à distribuição por Walsh.

Tendo o documento SGML e a especificação DSSSL do estilo era preciso um processador que percebesse as duas linguagens, fosse capaz de juntar os dois ficheiros e produzir o resultado final desejado. A escolha dessa ferramenta foi simples, o **jade** [jade] é o motor de DSSSL que suporta mais funcionalidades da linguagem (existem outros como o SENG escrito em Java mas o suporte da linguagem é muito parcial), e além disso não tem direitos comerciais, como vem sendo hábito no software desenvolvido por James Clark (é distribuído ao abrigo da licença da GNU).

Com o documento SGML e a especificação de estilo DSSSL, o jade pode converter o documento em RTF, TeX, MIF, HTML, texto ASCII, ou FOT (documento xml que descreve o grove final do processo de formatação, a *Flow Object Tree*). Como a essência do formato HTML é diferente dos outros, seriam necessárias duas especificações de estilo já previstas na distribuição das stylesheets. Houve que parametrizar cada uma delas.

A parametrização feita pelo autor para a versão papel foi a seguinte:

```
<!DOCTYPE style-sheet PUBLIC -
//James Clark//DTD DSSSL Style Sheet//EN" [
<!ENTITY dbstyle SYSTEM "..\sty-
le1.50\docbook\print\docbook.dsl" CDATA DSSSL>
]>

<style-sheet>
<style-specification use="docbook»
<style-specification-body>

(define %paper-type% "A4")
```

```
(define %default-title-end-punct% " ")
(define %two-side% #t)
(define %section-autolabel% #t)
(define %example-rules% #t)

(define biblio-citation-check #t)
(define biblio-number #f)

</style-specification-body>
</style-specification>
<external-specification id="docbook" document="dbstyle»
</style-sheet>
```

Como se pode ver, um grande número de parâmetros estão relacionados com a mancha de texto no papel.

A parametrização para HTML é ligeiramente diferente:

```
<!DOCTYPE style-sheet PUBLIC -
//James Clark//DTD DSSSL Style Sheet//EN" [
<!ENTITY dbstyle SYSTEM "../sty-
le1.44/docbook/html/docbook.dsl" CDATA DSSSL>
]>

<style-sheet>
<style-specification use="docbook»
<style-specification-body>

(define %default-title-end-punct% " ")
(define %section-autolabel% #t)
(define %example-rules% #t)

(define biblio-citation-check #t)
(define biblio-number #f)

(element emphasis
  (let ((func (if (attribute-string (normalize "ROLE"))
   (attribute-string (normalize "ROLE"))
   (normalize "normal"))))
    (cond
      ((equal? func "SCRIPT") ($italic-mono-seq$))
```

```
      ((equal? func "NORMAL") ($italic-seq$))
      (else (literal func)))))

</style-specification-body>
</style-specification>
<external-specification id="docbook" document="dbstyle»
</style-sheet>
```

As parametrizações relacionadas com o papel desapareceram. Podemos ver aqui uma redefinição de estilo para o elemento **EMPHASIS**. Está-se a usar este elemento para inserir letras num estilo matemático. Para esse efeito, o seu atributo **ROLE** toma um valor que indica o tipo de conteúdo do elemento. O estilo é portanto condicionado pelo conteúdo do atributo **ROLE**.

Para a versão papel, optou-se de início por gerar RTF. O resultado era directamente visível no MSWord Viewer, o que permitia trabalhar num só sistema operativo (o AuthorEditor só existe para Windows). As coisas corriam bem até se ter atingido o montante de 120 páginas. A partir daqui o MSWord Viewer começou a ter um comportamento não determinístico em relação às imagens, umas eram carregadas outras não (as imagens no formato RTF são mantidas externamente, no código há apenas um comando de inclusão da imagem; é da responsabilidade do software que irá abrir o documento, o carregamento das imagens). Com mais algumas páginas, as imagens simplesmente deixaram de ser carregadas o que impossibilitava a sua impressão e mesmo a disposição correcta do texto uma vez que o respectivo espaço não era reservado.

Tínhamos um problema grave, era preciso gerar outro formato para a impressão em papel. O avançado estado de escrita da tese não permitia grandes aventuras, e por uma questão de princípio era conveniente manter a produção em SGML e a formatação em DSSSL. A opção foi começar a gerar TeX. A partir do TeX era também possível gerar PDF o que nos deu dois caminhos à escolha:

1. Gerar TeX; converter as imagens para EPS; o resultado final para impressão seria o Postscript.
2. Gerar TeX; converter o TeX em PDF; converter as imagens para PNG; o resultado final para impressão seria o PDF (que permite também distribuir como edição electrónica - CDROM).

Como o resultado da conversão das imagens para EPS não foi satisfatório, ao contrário da conversão daquelas para PNG, optou-se pela segunda via.

O jade gera um formato TeX especial, que para ser processado precisa duma versão e duma configuração especial: o **jadetex** (que vem acompanhado do **pdfjadetex** que realiza a conversão posterior para PDF).

O jadetex é um processador de TeX específico que realiza a conversão pretendida muito bem, contudo a sua configuração é muito penosa e, ainda, preenchida com alguns passos obscuros.

Depois desta plataforma estar instalada, tudo começou a correr melhor, as imagens já eram carregadas (no caso do PDF elas são incluídas dentro do ficheiro final), mas as tabelas (apenas 3 ou 4) apareciam completamente desformatadas. Com a ajuda preciosa de um dos autores do jadetex, o Sebastian Rhatz, constatou-se que a versão do jade que eu utilizava era a mais pura mas para o caso precisava duma versão extendida desenvolvida por um grupo de utilizadores que decidiram levar o desenvolvimento do jade a sério e criaram o projecto de desenvolvimento **OpenJade** [OpenJade].

Depois de instalado (na versão 1.2.2), o circuito montado ficou a funcionar em pleno, permitindo gerar esta versão da tese.

A.4. A organização da tese em módulos

Dada a dimensão da tese e a variedade de formatos para cada imagem (cada imagem é mantida em GIF, PNG e WMF), o seu desenvolvimento dentro de um só ficheiro cedo deixou de ser viável. Assim, houve que estruturá-la em módulos e fazer corresponder cada módulo a um documento.

A tese foi dividida em capítulos, cada capítulo é um documento SGML. Cada imagem é mantida num ficheiro no sistema. Imagens do mesmo formato estão agrupadas numa colecção. Há um documento especial que é o elemento agregador, onde estão declarados os vários módulos, as várias colecções de imagens, e onde está especificado em que ordem os módulos são carregados e, no caso particular das imagens, qual a colecção que deve ser carregada (as colecções de imagens encontram-se em secções condicionais e só uma deverá estar activa.

A seguir apresenta-se um extracto desse documento comentado:

```
<!DOCTYPE BOOK PUBLIC -//OASIS//DTD DocBook V3.1//EN"
[

<!- O formato PNG para imagens não fazia parte
     do DTD, houve que declará-lo.               ->

<!NOTATION PNG SYSTEM "PNG»

<!- Declaração dos vários módulos ->

<!ENTITY header SYSTEM "header.sgm»
<!ENTITY prefacio SYSTEM "prefacio.sgm»
<!ENTITY chap2 SYSTEM "chap2-doc.sgm»
...
<!ENTITY ap-a SYSTEM "ap-a.sgm»
...
<!ENTITY biblio SYSTEM "biblio.sgm»

<!- Tratamento das letras usadas na matemática
     enquanto as entidades do ISOmscr não forem suportadas ->
<!ENTITY % my-script SYSTEM "script.ent»
%my-script;

<!- Tratamento das várias colecções de imagens ->
<!ENTITY % my-wmf SYSTEM "my-wmf.ent»
<!ENTITY % my-gif SYSTEM "my-gif.ent»
<!ENTITY % my-png SYSTEM "my-png.ent»

<!- Apenas uma é carregada: include ->
<!ENTITY % RTF "INCLUDE»
<!ENTITY % HTML "IGNORE»
<!ENTITY % TEX "IGNORE»
]>

<!- tese.sgm: o documento ->
<BOOK LANG="pt»
&header;
<TOC></TOC>
&prefacio;
&chap2;
...
```

```
&ap-a;
...
&biblio;
</BOOK>
```

Os restantes módulos são documentos SGML simples.

Para gerir todas as especificações envolvidas, DTD - cerca de 30 ficheiros, especificação DSSSL - cerca de 60 ficheiros, tese - cerca de 110 ficheiros, usaram-se identificadores públicos (acaba por ser um identificador abstracto). Este identificadores públicos são depois mapeados em identificadores do sistema; isto faz-se usando declarações próprias no ficheiro **catalog**. A vantagem é que podemos reutilizador várias vezes um módulo, carregando o seu conteúdo, referenciando-o por aquele identificador público. Se um dia decidirmos alterar a localização ou o ficheiro que está a ser usado para o módulo só será preciso alterar a respectiva declaração no **catalog**.

A título de exemplo apresenta-se o **catalog** principal da tese:

```
-Main Catalog file for my phd thesis -
-jcr 1999.11.12                      -

CATALOG "..\dtd\docbook.cat"
CATALOG "wmf.cat"
CATALOG "gif.cat"
CATALOG "eps.cat"
CATALOG "iso-ent.cat"

CATALOG "..\style1.50\docbook\catalog"
```

É fácil concluir que este catálogo é constituído apenas por referências a outros catálogos. Por exemplo, a primeira declaração indica onde está o catálogo para o DTD (cada versão do DTD traz um - para actualizar para uma versão mais actual basta-nos instalar a nova versão e alterar esta declaração para procurar o novo catálogo), a última indica qual o catálogo para a especificação DSSSL. As outras entradas carregam os meus catálogos das colecções de imagens.

A título de exemplo apresenta-se a seguir um excerto do catálogo para as imagens PNG:

```
- CHAP8 - Validacao com GAs - S4 -
PUBLIC -//jcr//ENTITIES S4: Arquitectura Concep-
tual 19991216//PNG" "../imagens/PDF/s4-arq-conceptual.png"
PUBLIC -//jcr//ENTITIES INES: Arquitectura Funcio-
nal 19991112//PNG" "../imagens/PDF/ines-arq.png"
PUBLIC -//jcr//ENTITIES S4: Arquitectura Funcio-
nal 19991217//PNG" "../imagens/PDF/s4-arq-func.png"
PUBLIC -//jcr//ENTITIES Arquitectura funcional dum editor es-
truturado 20000120//PNG" "../imagens/PDF/estrutura-editor.png"
```

Esta apresentação dos catálogos encerra a discussão da implementação modular da tese.

A.5. Resultados gerados

As diferentes versões da tese, nos dois formatos referidos obtêm-se invocando o jade:

PDF

Para gerar o PDF são necessários dois passos. No passo intermédio gera-se TeX e só depois se converte este para PDF:

```
jade -t tex -d jcr.dsl tese.sgm
pdfjadetex tese.tex
```

a opção **-t** indica qual o formato desejado para o resultado; a opção **-d** indica qual a especificação DSSSL a utilizar; por fim o terceiro elemento é nome do documento SGML que se quer processar.

HTML

A geração de HTML é mais fácil, basta invocar o jade e indicar que se quer transformar o documento SGML passado como parâmetro noutro documento SGML (lembrar que o HTML é SGML).

```
jade -t sgml -d jcr2.dsl tese.sgm
```

> Note que a especificação DSSSL é outra, correspondente à parametrização para HTML das folhas de estilo distribuídas. As imagens usadas nesta versão estão em formato GIF.

Termina aqui a redacção do "Making of My Thesis".

O objectivo deste apêndice foi mostrar que: muitas das ideias defendidas na tese estão aplicadas na sua edição electrónica; a tecnologia SGML é viável e funciona; se fosse adoptada em Universidades e outras instituições académicas, como formato obrigatório para a produção de teses, tornaria a distribuição e publicação destas mais fácil e consequentemente o seu conteúdo acessível a mais gente, bem como ajudaria a ajustar o DTD e até as ferramentas para a sua edição/formatação.

Apêndice B. Projectos desenvolvidos

Para se trabalhar e perceber bem o SGML é necessário estar envolvido em problemas reais. Só assim se consegue atingir um nível de compreensão do standard suficientemente alto para apontar defeitos e porpôr novas soluções. Desta necessidade de envolvimento surgiram uma série de projectos, uns grandes outros mais pequenos, dos quais se destacam os que a seguir se descrevem.

Os projectos encontram-se agrupados em dois grupos. O primeiro designado por "Publicação Electrónica"tinha o ponto de partida num formato digital não SGML. O segundo, designado por "Recuperação de edições esgotadas", tinha o seu ponto de partida numa versão em papel. Portanto, o tratamento dumas fontes e doutras foi diferente como a seguir se expõe.

B.1. Publicação Electrónica

Em 1989-90, foi desenvolvida, no Arquivo Distrital de Braga, uma linguagem de anotação para a edição dos livros que então se tentava publicar. De nome HiTex, por basear a sua sintaxe no TeX e usar aquele processador como ferramenta de formatação e composição final, foi usada na anotação de vários livros.

Apesar do HiTeX ser uma linguagem de anotação anterior ao SGML (pelo menos na altura não havia conhecimento da existência deste), seguia já alguns dos bons princípios do SGML. No entanto, na pressão final para a publicação daqueles livros muito código procedimental foi-lhes acrescentado, tronando hoje a sua alteração e reedição muito difícies. Surgiu então a ideia, em 1997, de recuperar aqueles textos convertendo-os em documentos SGML para posterior publicação na Internet e papel. É disso que tratam os três projectos que se descrevem a seguir.

B.1.1. Publicação na Internet das "Memórias Particulares de Inácio José Peixoto"

Como o título indica, o objectivo final foi a publicação na Internet do referido livro. Como ponto de partida havia as fontes em formato HiTeX.

O projecto seguiu as seguintes fases:

- Análise documental da obra e especificação do DTD.
- Eliminação das anotações HiTeX redundantes (lixo de formatação) - com o objectivo de facilitar o processamento seguinte.
- Especificação dos filtros para realizar a conversão HiTeX → DTD criado.
- Especificação dos filtros para realizar a conversão SGML → HTML, e geração dos respectivos índices (geral, antroponímico e toponímico).

Uma vez que a mão-de-obra era de alunos havia preocupações pedagógicas na realização do projecto. Foi por isso que não se utilizou um DTD existente e se desenvolveu um novo. A ideia era a de que os alunos cobrissem, o mais possível, todo o ciclo de desenvolvimento. No entanto, com o DTD e o documento anotado é sempre possível realizar a transformação do documento anotado noutro documento que respeite um novo DTD escolhido.

Para a construção dos filtros e conversores usaram-se duas bibliotecas Perl desenvolvidas para processar documentos SGML: *SGMLS.pm* e *NSGMLS.pl*.

Foi neste projecto que surgiu o primeiro problema de normalização. Havia que construir uma série de índices para permitir uma "navegação"inteligente no documento. No caso dos índices toponímico e antroponímico (haviam sido preparados à mão por historiadores), existiam muitos casos em que o nome no índice era diferente do nome no documento (o nome no índice seria o mais conhecido para o personagem em causa). Para estes casos aplicou-se a solução de normalização apresentada na Secção 7.1.2.

Os resultados visíveis deste projecto encontram-se disponíveis no *site* da Internet do Arquivo Distrital de Braga: *http://www.adb.pt*.

B.1.2. Publicação na Internet do "Index das Gavetas do Cabido da Sé de Braga"

Este projecto foi desenvolvido pela mesma equipe do projecto anterior e a estratégia foi semelhante. Os problemas encontrados foram os mesmos e as soluções seguidas também.

De momento, os resultados não se encontram disponíveis na Internet por que uma breve exposição permitiu detectar uma série de erros pelo que, os documentos estão em revisão.

B.1.3. Publicação na Internet do Livro "Ensaio Sobre as Minas de Joze Anastacio da Cunha"

Este projecto desenvolveu-se ao mesmo tempo que os outros dois. As metodologias aplicadas foram as mesmas. Surgiu no entanto um problema diferente: a matemática.

Joze Anastacio da Cunha foi um famoso engenheiro português que era muito requisitado para gerir a escavação de minas. Também era um bom construtor de armas e foram estas duas valências que o chamaram às cortes dos reis de França. Um dos documentos escritos por ele é este livro que acaba por ser um manual para a escavação de minas. Por isso e devido à formação do autor, o livro apresenta diversas fórmulas matemáticas de complexidade variada, num total de 208. Era aqui que residia o problema.

A colocação da matemática no papel é um problema que vem apaixonando pessoas da área da computação há algum tempo. A matemática não segue uma ordem linear como o texto, mas sim uma distribuição espacial a duas dimensões. Em SGML, o problema ainda não foi inteiramente resolvido. Há, no entanto, algum consenso na utilização da linguagem de anotação MathML para a representação da matemática. O MathML é um DTD definido a partir do TeX (o TeX ainda é o sistema de processamento que melhor trata a matemática), a sintaxe é a do SGML, os comandos são os que já existiam no TeX.

Tendo as fórmulas devidamente codificadas em MathML surgia agora o problema de como produzir um resultado final para dispobilizar na Internet (lembrar que o HTML também não tem capacidades para representar matemática). Na altura, devido a uma deslocação a uma conferência de SGML estabeleceu-se contacto com o grupo "*alphaworks*"da IBM. Eles estavam a desenvolver um "*plug-in*"para o Netscape, designado por *Techexplorer*, que permitia que, no meio duma página HTML, surgissem fórmulas matemáticas anotadas segundo o MathML. O *Techexplorer* desenhava a imagem correcta da fórmula em tempo de execução na janela do browser.

Adoptou-se então esta solução. Instalou-se o Techexplorer nas máquinas que deveriam ser usadas para visionar o livro e no tratamento de conteúdos usou-se MathML para a codificação das fórmulas matemáticas.

Apesar de tudo e uma vez que estávamos perante a versão inicial do Techexplorer só foi possível tratar 200 das 208 fórmulas. Motivo pelo qual o livro ainda não está

disponível na Internet.

Actualmente, já existe uma nova versão do Techexplorer bastante mais evoluída e, nos próximos tempos, tentar-se-á de novo ver se as restantes fórmulas se podem tratar.

B.2. Recuperação de edições esgotadas

Como já se disse, a diferença principal entre os projectos que a seguir se descre vem e os próximos, foi o ponto de partida. Nos projectos anteriores, tínhamos os documentos num formato e suporte digital. Nos que se apresentam a seguir apenas se possuía o documento em papel.

Tratavam-se de documentos com idade anterior à informatização do Arquivo pelo que haviam sido dactilografados. Mais grave ainda foi terem-se deixado esgotar os volumes então editados sem os reeditar ou mesmo convertê-los num formato digital.

O ponto de partido foi portanto, uma versão fotocopiada do documento original.

B.2.1. Recuperação do livro "Índice da gaveta das Cartas"

Este projecto teve como finalidade recuperar e reproduzir para vários formatos electrónicos um livro que já não é editado e se encontra esgotado, de nome *"Inventário das Cartas do Cabido de Braga"*, existente no Arquivo Distrital de Braga.

As etapas que constituíram o projecto foram:

- Análise documental e elaboração dum DTD para o documento em causa.
- Digitalização das páginas, recorrendo a um *scanner*.
- Conversão da imagem digitalizada em texto: para o efeito utilizou-se um software de OCR ("Online Character Recognition") que permitia, entre outros, converter a imagem num documento no formato RTF, que foi o escolhido.
- Conversão RTF → XML: foi necessário recorrer a um filtro desenvolvido em Omnimark por Rick Geimer para realizar esta conversão; depois de configurado,

o filtro permitiu a passagem do documento para um XML bastante pobre (a estrutura era basicamente constituída por parágrafos que tinham, porém, dois atributos, estilo e tamanho de letra, com os quais se conseguia determinar uma estrutura mais rica).

- Conversão XML → XML: coversão para o formato XML que está de acordo com o DTD definido na fase de análise (esta conversão só foi possível graças áqueles dois atributos; testando as combinações dos seus dois valores foi possível reconstruir uma estrutura com capítulos, secções, subsecções e parágrafos); o conversor foi desenvolvido em Omnimark.
- Anotação manual dos elementos pertencentes aos índices: para os elementos constantes nos índices, nomes de pessoas e lugares, foi necessário anotar o documento caso a caso, usou-se para o efeito o XMetal.
- Desenvolvimento dos conversores para as versões HTML e LaTeX (impressão em papel): utilizou-se de novo o Omnimark para o desenvolvimento destes conversores.

Os resultados deste projecto encontram-se no Arquivo. A versão HTML encontra-se já disponível na Internet: *http://www.adb.pt*.

B.2.2. Recuperação do livro "Inventário da gaveta das Visitas e Devassas"

Este projecto teve como finalidade recuperar e reproduzir para vários formatos electrónicos um livro que já não é editado e se encontra esgotado, de nome *"Inventário da gaveta das Visitas e Devassas"*, existente no Arquivo Distrital de Braga.

Este projecto seguiu a mesma linha de acção do anterior pelo que ficaremos por aqui na sua descrição.

Relativamente aos resultados, o livro ainda não se encontra disponível porque a equipe que trabalhou no projecto ao anotar os elementos dos índices deparou com muitas incorrecções semânticas o que motivou uma grande revisão dos conteúdos do livro que está a ser realizada pelos autores do livro. No entanto, o Arquivo está em posse de tudo o que é necessário para introduzir as correcções no documento XML final e gerar automaticamente as versões de distribuição.

B.3. Outros Projectos

O autor tem vindo a ser solicitado para trabalhar nalguns projectos reais às vezes como consultor.

Num desses contactos, surgiu o convite para integrar, juntamente com o supervisor, uma equipe do INESC que tem um projecto com o objectivo de elaborar uma proposta para o MPEG7 (espera-se que seja o formato de vídeo num futuro próximo).

Neste projecto, a responsabilidade do autor tem sido o desenvolvimento duma linguagem de anotação para a descrição de conteúdos: vídeo, imagem e texto.

O projecto teve orçamento do PRAXIS durante um ano para se melhorar e aprofundar as ideias iniciais pelo que, irá ser submetido de novo na próxima candidatura.

Apêndice C. Conversão de DTDs para Gramáticas

Um dos contributos desta tese foi a sistematização da conversão de um DTD SGML para uma Gramática de Atributos. A primeira aproximação a esta sistematização foi feita na tese de mestrado de Alda Lopes [Lop98]. A versão de então revelou-se insuficiente e com algumas lacunas quando aplicada aos casos de estudo. Houve, portanto, necessidade de refinar aquele primeiro estudo e proceder à correcção de vários problemas e a uma maior generalização do conjunto de regras de conversão. O resultado final apresenta-se neste apêndice e deve ser interpretado como um manual de instruções para a realização ou implementação duma conversão.

Algumas destas regras podem parecer simples e são-no, mas há algumas que não o são. A ideia deste estudo é a de ser a primeira versão duma especificação formal da conversão de DTDs em Gramáticas.

O leitor atento perceberá também facilmente que este estudo dá uma visão interior do SGML ajudando a perceber o porquê de certas características e a sua semântica.

As regras apresentadas encontram-se agrupadas por classes correspondendo cada classe a uma secção.

Assim temos um conjunto de regras gerais, que serão aplicadas a todas as declarações. Depois temos dois grupos de regras um para aplicar às declarações de elementos e outro para aplicar às declarações de atributos. A conversão da declaração dum elemento num conjunto de produções é um processo de cálculo recursivo: a expressão de conteúdo de um elemento pode ser composta por várias subexpressões aninhadas e às quais estão aplicados vários operadores com diferentes prioridades; o agrupamento e a prioridade dos operadores tem que ser captado na gramática. Este problema é semelhante ao do cálculo duma expressão aritmética onde é preciso ter em conta a precedência e a associatividade dos operadores pelo que a solução implementada anda muito próxima da dum calculador de expressões matemáticas só que o cálculo é o de uma gramática de atributos.

No que se segue, seja **Conv()** a função correspondente à aplicação recursiva das regras enunciadas.

C.1. Regras gerais

As regras gerais são as mais simples e as que deverão mais utilizadas.

C.1.1. Elemento genérico

Seja **elem** um elemento declarado no DTD, e **Content** a expressão de conteúdo que o define:

```
<!ELEMENT elem  Content>
```

Independentemente de ter atributos associados ou não, a correspondente produção gramatical é:

$$\texttt{elem} \rightarrow \texttt{elemAttList elemContent}$$

onde **elemContent** é o símbolo não-terminal que representa todo o seu conteúdo e **elemAttList** o símbolo não-terminal correspondente à declaração da lista de atributos de **elem**.

C.1.2. Elemento sem atributos

Se **elem** não tiver nenhum atributo declarado, **elemAttList** deriva na produção vazia:

$$\texttt{elemAttList} \rightarrow \epsilon$$

Esta é a situação inicial pois primeiro surge a declaração do elemento e só mais tarde,no DTD, surge a declaração dos atributos daquele elemento. Nessa altura, esta produção é reescrita de acordo com a regra aplicada e que tem a ver com o tipo de atributo.

Exemplo C-1. Elemento Titulo sem atributos

$$\texttt{Title} \rightarrow \texttt{TitleAttList TitleContent}$$
$$\texttt{TitleAttList} \rightarrow \epsilon$$

C.1.3. Elemento com conteúdo #PCDATA

Se o conteúdo dum elemento fôr atómico, do tipo **#PCDATA**, **elemContent** terá a seguinte derivação:

```
elemContent → textList

textList → text textList
         | ϵ

text → STR
```

Esta regra é assim apenas inicialmente. Há situações que irão provocar a sua reescrita (para completar as alternativas de **text**), como a inclusão de secções marcadas ou a definição de entidades no DTD (ver Secção C.4).

Exemplo C-2. Elemento com conteúdo #PCDATA

O conteúdo do elemento **Begin-date** está definido como **#PCDATA** no DTD do serviço noticioso, o que faz com que este seja derivável nas seguintes produções:

```
Begin-date → Begin-dateAttList Begin-dateContent

Begin-dateContent → textList

textList → text textList
         | ϵ

text → str
```

C.1.4. Elemento definido como um grupo - ()

Quando o conteúdo do elemento em definição não é atómico, **#PCDATA**, é definido por um grupo de elementos (um ou mais entre parentesis).

Nesse caso, o conjunto de produções em que deriva o conteúdo é:

```
elemContent → GrupoX

GrupoX → Conv(elemContent)
```

Em que **GrupoX** é um símbolo não-terminal introduzido pela regra e cujo nome é formado pela string **"Grupo"** concatenada com o valor dum contador interno **X** que logo é incrementado.

Às vezes, os grupos são singulares, **elemContent** é constituído apenas por um elemento. Neste caso, esta regra poderá ser abreviada fundindo as duas produções apresentadas.

C.2. Conversão de declarações de elementos

Se o conteúdo do elemento que se está a derivar fôr estruturado vai ser necessário analisar a estrutura desse conteúdo antes de se fazer a conversão.

Primeiro, é preciso determinar o operador de ocorrência: *zerone* para zero ou uma ocorrências (operador **?** do SGML); *zeron* para zero ou mais ocorrências (operador ***** do SGML); e *onen* para uma ou mais ocorrências (operador **+** do SGML). Se este não existir, passamos de imediato ao passo seguinte.

A seguir, analisa-se o operador de conexão: *sequencial (seq)* - os subelementos deverão aparecer todos pela ordem indicada; *alternativa (or)* - só um dos sublementos deverá aparecer.

Esta análise a dois passos vai-se repetir recursivamente para os sublementos até elementos atómicos serem atingidos (**#PCDATA**).

C.2.1. Elemento com estrutura singular

Se o a expressão de conteúdo fôr formada por apenas um elemento (**elem**). A respectiva regra de derivação é:

```
elemContent → elem
```

C.2.2. Operador de ocorrência zerone

No caso do sistema noticioso tínhamos:

```
<!ELEMENT News - - (Title, Begin-date, End-date?, Body?)>
```

O elemento **Body** pode aparecer zero ou uma vezes no conteúdo do elemento **News**, assim como o elemento **End-date**. Poderíamos tratar esta situação na gramática ao nível das derivações de **News**:

```
News → NewsAttList NewsContent

NewsContent → Grupo1

Grupo1 → Title Begin-date End-date Body
       | Title Begin-date End-date
       | Title Begin-date Body
       | Title Begin-date
```

Desta maneira, temos de trabalhar com permutações, o que torna o processo muito complicado. A situação simplifica-se introduzindo um símbolo não-terminal para cada elemento (ou grupo de elementos numa sub-expressão de conteúdo) que tiver um operador de ocorrência a ele aplicado.

Assim, sempre que uma sub-expressão (**SubExp**) fôr afectada de um operador de ocorrência **?**:

```
SubExp = A?
```

essa sub-expressão será denotada por um símbolo não-terminal de nome **GrupoX_01**, cujas produções para a sua derivação são:

```
GrupoX_01 → GrupoX
          | ϵ
```

```
GrupoX → Conv(A)
```

Onde, **GrupoX_01** corresponde à subexpressão inicial **A?**, e **GrupoX** é o símbolo não-terminal que irá derivar a subexpressão que está afectado pelo indicador de ocorrência (para a derivação de sub-expressões convencionou-se que os nomes dos símbolos não-terminais seriam formados pela palavra **Grupo** seguida do valor dum contador interno, que é incrementado cada vez que se gera um novo nome, seguidos dum indicador do operador de ocorrência: **01**,**0n**, ou **1n**).

Neste ponto, o processo de conversão aplicar-se-ia recursivamente à sub-expressão **A**, como se pode observar pela última produção.

Voltando ao nosso exemplo, a derivação de **News** será:

```
News → NewsAttList NewsContent

NewsContent → Title Begin-date Grupo1_01 Grupo2_01

Grupo1_01 → Grupo1
          | ϵ

Grupo1 → End-date

End-date → End-dateArrList End-dateContent

Grupo2_01 → Grupo2
          | ϵ

Grupo2 → Body

Body → BodyAttList BodyContent
```

C.2.3. Operador de ocorrência zeron

Novamente do caso do sistema noticioso, podemos extrair outro exemplo para ilustrar esta regra (vamos alterá-lo ligeiramente e supôr que Body é composto por zero ou mais parágrafos):

```
<!ELEMENT Body - - (Para)*>
```

Aqui, o operador de ocorrência implica iteração o que vai traduzir em recursividade na gramática.

Assim, sempre que uma sub-expressão (**SubExp**) fôr afectada de um operador de ocorrência *****:

```
SubExp = A*
```

essa sub-expressão será denotada por um símbolo não-terminal de nome **GrupoX_0n**, que deriva segundo as produções:

```
GrupoX_0n → GrupoX  GrupoX_0n
          | ϵ

GrupoX → Conv(A)
```

A conversão da definição do elemento **Body** corresponde então ao conjunto seguinte de produções:

```
Body → Grupo3_0n

Grupo3_0n → Grupo3 Grupo3_0n
          | ϵ

Grupo3 → Para
```

C.2.4. Operador de ocorrência onen

Vamos usar o exemplo da regra anterior substituindo apenas o operador de ocorrência:

```
<!ELEMENT Body - - (Para)+>
```

A situação é tratada de forma idêntica à da regra anterior, sendo neste caso a sub-expressão **A+** denotada pelo símbolo não-terminal **GrupoX_1n** que não pode derivar em vazio:

```
GrupoX_1n → GrupoX  GrupoX_1n
          | GrupoX

GrupoX → Conv(A)
```

Donde, para o exemplo em causa se terá:

```
Body → Grupo3_1n

Grupo3_1n → Grupo3 Grupo3_1n
          | Grupo3

Grupo3 → Para
```

Como foi dito, depois da análise do operador de ocorrência faz-se a análise do operador de conexão que pode ser *seq* ou *or*; sejam, *op1, ..., opn,* os elementos (operandos) abrangidos por um desses operadores. Duas situações poderão ocorrer. As regras seguintes foram criadas para tratar estas situações.

C.2.5. Operador de conexão seq (sequência)

Os operandos duma sequência têm que ser derivados na ordem em que se encontram na sequência. Assim, é fácil ver que a produção correspondente é:

```
elemContent → Conv(op1) ... Conv(opn)
```

Exemplo C-3. Operador de conexão seq

Como já visto no nosso sistema noticioso, o elemento **News** é composto por uma sequência de elementos, o que dá origem à seguinte produção:

```
NewsContent → Title Begin-date Grupo_01 Grupo_02
```

C.2.6. Operador de conexão or (alternativa)

Os seus operandos são alternativas, assim só um corresponderá ao conteúdo do elemento.

As produções correspondentes serão:

```
elemContent → Conv(op1)
            | ...
            | Conv(opn)
```

C.3. Regras para os atributos

Cada um dos atributos no DTD vai corresponder a um símbolo não-terminal na gramática de atributos.

O SGML permite definir atributos de vários tipos. Apresenta-se, primeiro, uma regra geral que deverá ser aplicada sempre, independentemente do tipo do atributo, e a seguir, as regras que permitem converter os tipos mais usuais de atributos.

C.3.1. Elemento com atributos

Se **elem** tem atributos, sejam eles **att1**, ..., **attn**.

A sua declaração no DTD seria:

```
<!ATTLIST elem att1 att1-type att1-def-value
              ...
              attn attn-type attn-def-value>
```

A produção principal correspondente à regra geral, é:

```
elemAttList → Att1 ... Attn
```

onde **Atti** é o símbolo não-terminal introduzido para representar o atributo **atti**.

att1-type, ..., attn-type representam o tipo de cada um dos atributos: *enumerado, CDATA, NUMERICAL, ID, IDREF, IDREFS, ENTITY,* e outros que já não se

utilizam. Na declaração de um atributo também podemos associar-lhe um valor por omissão, representado acima por **atti-def-value**, que deve ser tido em conta sempre que o atributo não fôr instanciado pelo utilizador. Os valores possíveis para este são os definidos na tabela abaixo: **#FIXED** seguido de uma string, o que significa que se o atributo fôr instanciado terá que ser com um valor igual a string; um dos valores pertencentes a um tipo enumerado definido para esse atributo; ou uma das palavras-chave seguintes:

Tabela C-1. Valores por omissão de Atributos

#IMPLIED	o valor poderá não ser instanciado, o atributo é opcional.
#REQUIRED	o utilizador terá obrigatoriamente que instanciar o atributo, este é obrigatório
#FIXED	o valor do atributo tem que aparecer a seguir à palavra-chave (ver Secção 7.1, Exemplo 7-3) significando que se o atributo fôr instanciado terá que ser com um valor igual ao declarado, caso não seja instanciado o sistema assumirá esse valor.
#CURRENT	o valor do atributo é herdado da última instância do mesmo elemento onde o valor do atributo foi instanciado (segue-se a ordem ascendente na árvore de elementos)
#CONREF	o valor é usado para referências externas (cada vez menos utilizado, não irá ser considerado nas regras que se seguem)
valor dum tipo enumerado	um dos valores pertencentes ao tipo enumerado definido para esse atributo.

O valor por omissão levanta-nos um problema: não é possível derivá-lo da gramática, ou seja, o comportamento deste item é essencialmente semântico - "se o atributo não estiver instanciado, então use-se este valor". Como tal, iremos recorrer

a equações sobre atributos (da gramática de atributos) para o modelar, como se poderá ver nas regras seguintes.

As restantes produções de cada um dos atributos dependem dos seus tipos e valores por omissão. Serão tratadas nas regras que se seguem.

Exemplo C-4. Elemento News com atributos

No nosso caso de estudo do serviço noticioso, o elemento *News* tinha dois atributos: *Type* e *Subject*. Assim, as produções correspondentes são:

```
News → NewsAttList NewsContent
NewsAttList → Type Subject
```

O conteúdo do elemento pode ser estruturado ou texto.

C.3.2. Atributo do tipo enumerado

Se um atributo **att** do elemento **X** fôr declarado da seguinte maneira:

```
<!ATTLIST X att (att-val1 | ... | att-valn) att-vali>
```

é do tipo enumerado: o seu tipo está definido como a lista *att-val1, ..., att-valn,*, e o seu valor por omissão é *att-vali* com $0<i<n$. As produções correspondentes são:

```
att → att-val1
    | ...
    | att-valn
    | ϵ
```

Os símbolos *att-val1, ..., att-valn* são símbolos terminais da gramática, correspondem a tokens/palavras-chave fixos; ϵ corresponde à string vazia e esta derivação corresponde à inexistência do atributo na instância documental (a situação em que o utilizador não instancia o atributo), e onde deverá ser assumido, portanto, o valor por omissão.

Assim as produções anteriores têm de ser enriquecidas com as seguintes equações sobre o atributo gramatical **valor**, que irá guardar o valor do atributo SGML **att**:

```
att → att-val1
        valor = f(att-val1)
    | ...
        ...
    | att-valn
        valor = f(att-valn)
    | ϵ
        valor = f(att-vali)
```

Aqui e no resto das regras, a função **f** representa a conversão de tipos necessária para passar o valor devolvido pelo analisador léxico (tipo=string) para o valor pretendido no atributo semântico.

Exemplo C-5. Atributo do tipo enumerado

No nosso sistema noticioso, o elemento **News** tem um atributo de nome **Type** do tipo enumerado, com valores **Event** ou **Novelty**, o que dá origem às seguintes produções:

```
Type → event
         valor = "Event"
     | novelty
         valor = "Novelty"
```

C.3.3. Atributo do tipo CDATA

Se um atributo fôr do tipo **CDATA**, significa que tem um conteúdo textual, e a produção correspondente é:

```
att → str
        valor = f(str)
```

O símbolo **str** é um símbolo terminal da gramática resultante.

C.3.4. Atributo do tipo NUMERICAL

Se um atributo fôr do tipo **NUMERICAL**, significa que tem um conteúdo numérico e a produção correspondente é:

```
att → number
        valor = f(number)
```

O símbolo **number** é um símbolo terminal da gramática resultante.

C.3.5. Atributo do tipo ID

Se um atributo fôr do tipo **ID**, significa que tem um conteúdo textual e uma semântica associada: o valor do atributo, instanciado pelo utilizador, tem que ser único, i.e., não pode haver no documento mais nenhum atributo deste tipo instanciado com o mesmo valor. A produção e semântica correspondentes são:

```
att → ident
       if( exists(ident, Tabid) )
         erro = "Identificador já existente!"
       else insert(ident, Tabid)
       valor = f(ident)
```

C.3.6. Atributo do tipo IDREF

Se um atributo fôr do tipo **IDREF**, significa que tem um conteúdo textual e uma semântica associada, que é o inverso da situação anterior: o valor do atributo, instanciado pelo utilizador, tem que existir na tabela de identificadores, tem que estar declarado num atributo do tipo **ID** algures no documento. A produção e a semântica correspondentes definem-se como:

```
att → ident
        if( !exists(ident, Tabid) )
          erro = "Identificador inexistente!"
```

```
        valor = f(ident)
```

C.3.7. Atributo do tipo ENTITY

Se um atributo fôr do tipo **ENTITY**, significa que tem um conteúdo textual e uma semântica associada: o valor do atributo, instanciado pelo utilizador, tem que ser o identificador de uma entidade existente na tabela de entidades, tem que estar declarado numa declaração de entidade no DTD (Secção C.4). A produção e a semântica correspondentes definem-se como:

```
att → ent-ident
        if( !exists(ent-ident, Tab-entidade) )
          erro = "Entidade inexistente!"
        valor = f(ent-ident)
```

C.3.8. Atributo definido com o valor #IMPLIED

Se um atributo fôr definido com o valor por omissão **#IMPLIED**, significa que é opcional, e as produções correspondentes são:

```
att → att-val
    | ϵ
```

A derivação pela string vazia acresce a uma das conversões anteriores dependendo do tipo. A produção att → att-val representa a produção resultante da conversão do tipo de atributo; ao juntarmos a conversão do valor por omissão vamos acrescentar produções à derivação de att e, nalguns casos, alguns atributos e respectivas equações de cálculo.

C.3.9. Atributo definido com o valor #REQUIRED

Se um atributo fôr definido com o valor por omissão **#REQUIRED**, significa que é obrigatório, e as produções correspondentes são:

```
att → att-val
```

```
              | ϵ
                  erro = "Atributo obrigatório!"
```

A derivação pela string vazia acresce a uma das conversões anteriores dependendo do tipo.

C.3.10. Atributo definido com o valor #FIXED

Se um atributo fôr definido com o valor por omissão **#FIXED**, significa que o seu valor foi fixo na declaração. O utilizador poderá sempre instanciá-lo mas o valor terá que ser igual ao fornecido na declaração do atributo no DTD. As produções e a semântica correspondentes definem-se como:

```
          att → att-val
                  if(f(att-val) != Tab-attr(att).valor)
                    erro = "O valor do atribu-
to não é o esperado!"
                  valor = Tab-attr(att).valor

              | ϵ
                  valor = Tab-attr(att).valor
```

A derivação pela string vazia e as condições de contexto acrescem a uma das conversões anteriores dependendo do tipo.

C.3.11. Atributo definido com o valor #CURRENT

Este valor por omissão só se utiliza para atributos pertencentes a elementos que podem aninhar-se. Por exemplo, um documento pode construir a sua hierarquia de secções e subsecções à custa de apenas um elemento secção ao qual, no DTD, se deu permissão para aninhar recursivamente. Assim, uma secção que ocorra dentro de outra secção é uma subsecção daquela ou uma secção de nível dois.

Este tipo de elementos tem, normalmente, atributos, que se forem declarados com o valor por omissão **#CURRENT** não precisam de estar instanciados, nesse caso, o atributo herdará o seu valor da última instanciação desse atributo percorrendo os elementos de baixo para cima até encontrar um que o tenha instanciado.

As produções e a semântica correspondentes definem-se como:

```
att → att-val
          valor = f(att-val)
    | ϵ
          valor = Procura(att, ASAD)
```

A função **Procura** vai tentar calcular o valor do atributo percorrendo a árvore de sintaxe abstracta decorada calculada até ao momento, em ordem ascendente.

A derivação pela string vazia e as condições de contexto acrescem a uma das conversões anteriores dependendo do tipo.

C.4. Regras de conversão para entidades

Sempre que uma entidade é declarada no DTD, o utilizador pode utilizá-la no texto (como uma abreviatura ou inclusão de uma imagem, por exemplo). Portanto quando uma ou mais entidades estiverem definidas no DTD os elementos de conteúdo textual passam a poder ter um conteúdo misto (texto onde, em qualquer lugar, podem aparecer referências a entidades).

C.4.1. Entidades

Sejam *ent1, ..., entn,* entidades definidas no DTD. Então, todos os elementos de conteúdo textual terão de permitir referências a qualquer uma daquelas entidades no seu conteúdo.

Por conseguinte, a regra apresentada em Secção C.1.3 para conteúdos do tipo **#PCDATA** teria de ser completada da seguinte maneira:

```
elemContent → textList

textList → text textList
         | ϵ

text → str
```

```
| ent1
| ...
| entn
```

Termina aqui a lista de regras de conversão. Estas regras encontram-se implementadas no sistema S4, descrito em detalhe num dos últimos capítulos (Secção 9.2).

C.5. Uma conversão passo a passo

No capítulo onde se discutiu a semântica dinâmica (Capítulo 6), foi introduzido um documento SGML que continha o seguinte DTD.

```
<!DOCTYPE RECEITAS [
<!ELEMENT RECEITAS    (TITULO,RECEITA*)>
<!ELEMENT TITULO      (#PCDATA)>
<!ELEMENT RECEITA     (TITULO,INGREDIENTE+)>
<!ELEMENT INGREDIENTE (#PCDATA)>
<!ATTLIST RECEITA
                      ORIGEM CDATA #IMPLIED>
]>
```

Agora, vamos seguir a sua conversão numa gramática de atributos, passo a passo, através da aplicação sistemática das regras definidas ao DTD:

Exemplo C-6. Uma conversão: DTD → GA

Para cada declaração do DTD apresentam-se as produções geradas sistematicamente:

Receitas (Titulo,Receita*)

$$\texttt{Receitas} \rightarrow \texttt{ReceitasAttList ReceitasContent}$$

$$\texttt{ReceitasAttList} \rightarrow \epsilon$$

```
ReceitasContent → Grupo1

Grupo1 → Titulo Grupo2_0n

Grupo2_0n → ϵ
          | Grupo2 Grupo2_0n

Grupo2 → Receita
```

Titulo (#PCDATA)

```
Titulo → TituloAttList TituloContent

TituloAttList → ϵ
TituloContent → textList

textList → text textList
         | ϵ

text → str
```

Receita (Titulo,Ingredente+)

```
Receita → ReceitaAttList ReceitaContent

ReceitaAttList → ϵ

ReceitaContent → Titulo Grupo3_1n

Grupo3_1n → Grupo3
          | Grupo3 Grupo3_1n

Grupo3 → Ingrediente
```

Ingrediente (#PCDATA)

```
Ingrediente → IngredienteAttList IngredienteContent
```

```
IngredienteAttList → ϵ

IngredienteContent → textList
```

ATTLIST RECEITA ORIGEM CDATA #IMPLIED

```
ReceitaAttList → Origem

Origem → str
       | ϵ
```

Com este exemplo, fica concluído este apêndice sobre a conversão de DTDs em Gramáticas de Atributos.

Apêndice D. Futuros standards relacionados com SGML

Desde o início desta tese já se passaram quatro anos. E foram quatro anos bem efervescentes. A indústria de software acordou para o SGML e para os standards relacionados com a estruturação de conteúdos (normalmente, para a internet). Nos últimos dois anos, tem havido uma explosão de ideias, os interesses multiplicaram-se bem como as pessoas neles envolvidas.

Este apêndice surge, ao terminar a escrita da dissertação, para fazer o ponto da situação actual. Será fácil concluir que, apesar da proliferação de ideias e de propostas para novos standards, os conceitos são os mesmos, a filosofia SGML está lá: formato neutro, estruturação de conteúdos, independência relativamente à apresentação.

A lista de especificações SGML bem como o número de propostas de novos standards submetidas ao W3C é enorme. Aqui, vamos focar as mais relevantes e aqueles para as quais se prevê um maior impacto e desenvolvimento, que andam indubitavelmente a gravitar em torno do XML

As especificações que se descrevem estão ainda em diferentes estágios de desenvolvimento (na sua maioria anda inacabadas), podem, por isso, ser alteradas. Isto não implica que não estejam a ser utilizadas; muito pelo contrário, há especificações ainda no primeiro nível de desenvolvimento já implementadas.

Actualmente, o W3C considera quatro níveis de desenvolvimento para as propostas de normalização que lhe são submetidas. Apresenta-se a seguir uma lista dos níveis de desenvolvimento e correspondentes especificações.

Recomendação

Significa que a especificação está estável, contribui para uma evolução da tecnologia e é suportada pelos membros do W3C que apoiam a sua aplicação.

- Extensible Markup Language (XML) 1.0
- Document Object Model (DOM) Level 1
- Cascading Style Sheets Level 1 (CSS1)

- Cascading Style Sheets Level 2 (CSS2)
- NameSpaces in XML (XML Namespace)

Recomendação Proposta

Significa que a especificação está a ser revista pelos membros do W3C.

- Neste momento não há nenhuma relacionada com o XML.

Rascunho

Significa que a especificação pode ser alterada em qualquer altura, substituída ou simplesmente posta de lado; este tipo de material não deve ser usado como referência podendo, no entanto, ser citado como "trabalho em desenvolvimento".

- Extensible Stylesheet Language (XSL)
- XML Linking Language (XLL)
- XML Pointer Language (XPointer)

Notas

Significa que se trata de uma especificação submetida ao W3C pelo público, ou por algum membro; não foi submetida num "Call for proposals"; por ter interesse ou qualidade é publicada e mantida para discussão.

- XML Query Language (XQL)
- XML Data (XML-Data)
- Schema for Object-oriented XML (SOX)
- Vector Markup Language (VML)
- Simplified Markup Language (SML)

No tabela seguinte, sintetiza-se o domínio de acção das propostas mais relevantes:

Tabela D-1. Domínios de acção

Especificação	Estilo	Transformação	Selecção
SGML	DSSSL	DOM	XQL
XML	XSL	XSL	XLL
SML	CSS1	SOX	XPointer
	CSS2		

A utilização combinada das várias linguagens é possível: pode-se montar um sistema escolhendo uma proposta de cada categoria tendo o cuidado de verificar se se está a escolher a mais adequada ao fim em causa. O XML representa a linguagem de anotação mais versátil podendo combinar-se com qualquer uma das outras categorias. O SGML também se poderá combinar com algumas das outras (com o DSSSL é o mais usual) mas com algumas adaptações. O SML, de momento não passa de uma ideia, de certa maneira retrógada, como iremos ver mais à frente.

Algumas destas propostas são explicadas a seguir.

D.1. Extensible Markup Language (XML) 1.0

E então surgiu o XML... E surge a pergunta: porquê XML e não SGML? Sim, porque o XML é apresentado como um subconjunto do SGML. À partida parece que se vai perder algo adoptando um e não o outro.

A resposta encontra-se na história e desenvolvimento da informática. O SGML surgiu já há mais de 10 anos e teve de conviver com os condicionalismos da época o que se traduz actualmente num pequeno subconjunto de características que deixaram de fazer sentido e que apenas atrapalham.

A linguagem de anotação XML foi definida pelo W3C [http://www.w3c.org], para a produção de conteúdos estruturados para a Internet.

As próximas secções darão resposta à questão que fica agora no ar: Porquê mais um standard para a Internet?

D.1.1. O Passado

O XML não é mais do que um subconjunto do SGML (Capítulo 4).

O SGML e o XML são mais do que linguagens de anotação, são meta-linguagens; permitem a definição de novas linguagens. Por outro lado, o HTML, a linguagem de anotação mais conhecida para a produção de páginas para a Internet, está definida em SGML. É uma linguagem concreta, não é possível estendê-la a não ser que se altere a sua definição inicial (o que a transformaria noutra linguagem).

O XML não tem esta limitação. Sendo uma meta-linguagem, permite em qualquer momento o acrescentar de novos elementos à linguagem. Na prática, isto traduz-se numa linguagem aberta que nos permite especificar qualquer coisa com o nível de detalhe que pretendermos.

Neste momento, o HTML continua a ser a linguagem mais utilizada na Internet mas, irá ser gradualmente substituído pelo XML.

D.1.2. Porquê XML?

O crescimento espantoso da Internet, nomeadamente do serviço WWW, nos últimos anos, é um facto que se deve principalmente à possibilidade de distribuir, facilmente e a baixos custos, informação e aplicações, a qualquer utilizador, em qualquer parte do mundo. À medida que a informação que se coloca na Internet se vai tornando cada vez mais complexa, e o número de utilizadores vai crescendo, as limitações das tecnologias e standards actuais vão-se tornando cada vez mais evidentes.

A limitação na construção de páginas WWW deve-se principalmente ao facto do HTML possuir apenas um conjunto fixo e pré-definido de etiquetas com o qual se pode definir a estrutura e a aparência duma página WWW. Foi esta limitação, e a impossibilidade de de criar extensões à linguagem que tornou a criação dum novo standard desejável. Nalgumas aplicações, principalmente onde uma publicação electrónica de alta qualidade e desempenho é o objectivo a atingir, tem-se vindo a adoptar o SGML para a produção e manutenção de conteúdos; as páginas WWW são depois geradas automaticamente, partindo destes conteúdos. No entanto, o SGML é complexo e de difícil aprendizagem o que fará com que seja apenas utilizado por comunidades que possuam um elevado nível técnico, e portanto, não tenha uma grande aceitação por parte dos utilizadores normais.

Nos últimos tempos, as aplicações distribuídas em geral, vieram realçar a falta dum standard para a transferência de dados estruturados. Apesar dos esforços e do

dinheiro investido esse standard não surgiu. Os problemas surgem quando diferentes aplicções querem interagir; normalmente, cada uma tem o seu próprio formato para comunicar dados que é diferente do da outra.

Para resolver estes problemas, foi criado o XML. O XML oferece um método estruturado e consistente para descrever e transferir informação. A melhor característica que possui, herdada do SGML, é a separação do formato visual da informação propriamente dita. Isto faz com que o XML seja a linguagem ideal para a produção de conteúdos textuais (inependência de plataformas de hardware e software, longevidade, ...). Duas aplicações XML podem enviar e receber informação livremente sem preocupações com o formato dessa informação, a informação contida num documento XML auto-descreve-se.

O XML foi concebido tendo uma série de objectivos em vista:

- Deve ser directamente utilisável na Internet - os utilisadores devem ser capazes de ver páginas XML da mesma maneira que vêem páginas HTML.
- Deve suportar uma série de aplicações: editores, browsers, sistemas de gestão de bases de dados, ... No entanto, o principal objectivo é a produção de conteúdos estruturados para a Internet.
- Deve ser compatível com o SGML - já existem muitos conteúdos em SGML e para quem os produziu a compatibilidade entre os dois é crítica.
- A escrita de programas para processar documentos XML deve ser simples.
- O número de catacterísticas opcionais deve ser reduzido a um mínimo, pois quantas mais houver mais problemas de compatibilidade poderão surgir.

D.1.3. XML: características

Como já foi dito, o XML acaba por ser um subconjunto do SGML. Uma vez que o SGML já foi detalhadamente descrito nesta tese, podemos descrever o XML enunciando quais as características do SGML que foram deixadas de fora.

Assim, vamos ter simplificações ao nível das declarações no DTD e da funcionalidade esperada.

Ao nível do DTD foram retiradas as seguintes características:

- Inclusões e exclusões.
- O operador **&**.
- Ao nível de entidades apenas podemos ter um nível de indireccionamento.

  As entidades do tipo caracter não fazem parte de nenhuma distribuição uma vez que foi adoptado o Unicode como conjunto de caracteres base [Unicode].

A diferença mais marcante entre o SGML e o XML é a importância dada à validação do documento com o DTD, como se pode ver na próxima secção.

D.1.3.1. Válido versus Bem-Estruturado

Numa linguagem, qualquer que ela seja, precisamos de ter uma entidade que verifique se um dado documento escrito nessa linguagem está de acordo com a gramática e especificações dessa linguagem. No nosso caso, o DTD corresponde à gramática e o parser ao verificador.

No entanto, o XML foi pensado para suportar aplicações "Web"e, em muitos casos, o DTD e as respectivas verificações podem tornar-se um fardo bem pesado. Então criaram-se dois patamares de validação: o documento válido e o documento bem-formado. Um documento diz-se válido se tiver um DTD associado e se o texto do documento está de acordo com as especificações no DTD; um documento diz-se bem-formado se obedecer às seguintes condições:

- Tiver um ou mais elementos.
- Há apenas um elemento (o elemento raíz) para o qual nem a anotação de início nem a anotação de fim estão dentro de qualquer outro elemento.
- Todas as outras anotações têm que estar aninhadas correctamente.
- Todas as entidades usadas no documento têm de estar definidas em XML ou no DTD.

Podemos ver agora dois exemplos: o primeiro dum documento bem estruturado e o segundo do mesmo documento mas agora com o DTD associado ilustrando o conceito de documento válido.

Exemplo D-1. Um documento bem estruturado

```
<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>
```

Exemplo D-2. Um documento válido

```
<!DOCTYPE RECEITAS [
<!ELEMENT RECEITAS    (TITULO,RECEITA*)>
<!ELEMENT TITULO      (#PCDATA)>
<!ELEMENT RECEITA     (TITULO,INGREDIENTE*)>
<!ELEMENT INGREDIENTE (#PCDATA)>
<!ATTLIST RECEITA ORIGEM CDATA #IMPLIED>
]>
<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>
```

Terminamos a apresentação do XML com um pequeno conjunto de regras que permitem transformar qualquer documento SGML num documento XML (lembrar que em XML não necessitamos do DTD pelo que basta que as anotações estejam de

acordo com as regras do XML para que o documento possa ser considerado um documento XML):

- Acrescentar no início do documento a seguinte instrução de processamento:

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

- Fechar todas as anotações iniciadas (não há anotações opcionais - o SGML permitia a ausência de anotações nalguns casos).

  Todos os elementos vazios passam de:

  ```
  "...discutido em <Referencia ci-
  teid="plima»</Referencia> ..."
  ```

  a

  ```
  "...discutido em <Referencia citeid="plima"/> ..."
  ```

  O que irá facilitar muito a vida ao parser (ao encontrar "/>", sabe que o elemento corrente termina ali).

- Todos os valores de atributos devem estar dentro de aspas.

D.2. Document Object Model (DOM)

Esta especificação define uma interface e uma plataforma neutras que irão permitir que programas e scripts acedam e alterem o conteúdo, a estrutura e o estilo dos documentos, duma forma normalizada. Isto permitirá uma maior portabilidade de programas e scripts.

Apesar do seu nome, o DOM não é um modelo orientado a objectos e sim, uma maneira independente de declarar interfaces e objectos para manipulação de documentos XML e HTML.

Vão ser disponibilizados três métodos para utilizar o DOM para aceder a documentos XML:

- Utilizando Javascript ou Vbscript numa página Web.
- Utilizando uma aplicação tipo "plug-in", ou ActiveX, que acedem ao documento através dum browser.
- Utilizando um parser XML externo que implementa o DOM.

Quer o Netscape Navigator, quer o Internet Explorer, suportam o DOM, mas com várias alterações não normalizadas. No entanto, ficou estabelecido que nas próximas versões dum e doutro, o DOM suportado seria o standard.

D.3. CSS e XSL

Neste momento, há duas linguagens de especificação de estilo para documentos XML em desenvolvimento: Cascading Style Sheets (CSS) e Extensible Stylesheet Language (XSL). Ambas serão descritas em mais detalhe nas próximas secções.

Neste momento, uma questão poderia ser levantada: porquê desenvolver duas linguagens e não apenas uma? O quadro seguinte fornece a resposta.

Tabela D-2. CSS versus XSL

	CSS	**XSL**
Pode ser usada com HTML?	sim	não
Pode ser usada com XML?	sim	sim
Pode transformar documentos?	não	sim
Sintaxe	CSS	XSL

Da análise do quadro, pode-se concluir que o CSS pode ser usado para formatar documentos XML e HTML, no entanto, não pode ser usado para transformar documentos. Por outro lado, o XSL apenas pode ser usado para formatar XML e pode transformar documentos. Podemos agora perguntar: quando usar CSS e quando usar XSL?

Sempre que o documento final fôr uma versão estilizada do documento original, devemos usar o CSS. Quando o documento final resultar duma transformação do

documento original (construção de índices, listas de nomes, ...), deve ser usado o XSL.

D.3.1. Cascading Style Sheets (CSS)

O CSS é uma linguagem declarativa muito simples que permite, a autores e utilizadores, associar informação de estilo a documentos estruturados XML ou HTML.

O estilo de um elemento especifica-se associando-lhe propriedades e valores.

Exemplo D-3. Especificação de estilo para um elemento

```
H1 {
     font-size: 16pt;
     font-weight: bold;
     color: blue;
   }
```

Neste exemplo, está-se a declarar que os elementos H1 devem ter o texto em 16 pontos, letra carregada e de côr azul.

As outras propriedades do CSS permitem especificar tudo desde o tamanho de letra dum parágrafo até margens, espaçamento de linhas e texturas de fundo.

O CSS tem dois níveis de especificação suportados em duas implementações diferentes:

CSS Level 1 (CSS1)

> O CSS1 é uma linguagem de fácil leitura e permite expressar o estilo numa terminologia comum a sistemas normais de publicação. Como o nome indica, várias especificações de estilo podem ser aninhadas para produzir o resultado final.

CSS Level 2 (CSS2)

O CSS2 define um nível de abstracção acima do CSS1 - dá um maior controle ao utilizador na disposição dos objectos na página. Com poucas excepções, todas as especificações CSS1 são especificações CSS2. Na secção seguinte descreve-se melhor o CSS2.

D.3.1.1. Cascading Style Sheets Level 2 (CSS2)

O CSS pode ser usado com qualquer documento estruturado. Vamos ver, por exemplo, a sua aplicação ao XML.

Exemplo D-4. Documento XML

```
  <?xml version="1.0"?>
  <CURSO>
    <TITULO>XML, uma promessa?... </TITULO>
    <AUTOR>José Carlos Ramalho</AUTOR>
    <RESUMO>
      <PARA>Durante este curso, faremos uma travessia do uni-
verso <ACRON>XML</ACRON> ... </PARA>
      ...
    </RESUMO>
    ...
  </CURSO>
```

Para formatar este documento recorrendo ao CSS2 temos de fazer duas coisas:

- Criar uma especificação de estilo
- Associar essa especificação de estilo ao documento XML

D.3.1.1.1. Especificação de Estilo

A especificação de estilo é simplesmente um ficheiro de texto com extensão ".css". Para este exemplo criou-se o ficheiro "curso.css".

Exemplo D-5. curso.css

```
ACRON {display: inline}
CURSO, TITULO, AUTOR, RESUMO, PARA {display: block}
TITULO {font-size: 16pt}
AUTOR {font-style: italic}
CURSO, TITULO, AUTOR, RESUMO, PARA {margin: 2cm}
```

As primeiras duas linhas especificam se os elementos são "inline"(devem ser colocados na mesma linha que os caracteres que os precedem e antecedem), ou "block"(há uma quebra de linha no início do elemento e outra no fim).

As outras linhas alteram o valor de algumas propriedades dos elementos a que se referem (tamanho de letra, postura, margens).

D.3.1.1.2. Associar uma Especificação de Estilo

Neste momento, a maneira de associar uma especificação de estilo a um documento XML é inserir no início do documento uma instrução de processamento com um determinado formato:

```
<?xml version="1.0"?>
<?xml:stylesheet type="text/css" href="curso.css"?>
... Resto do documento ...
```

D.3.2. Extensible Stylesheet Language (XSL)

O CSS é uma boa opção para a especificação do estilo sempre que os elementos (parágrafos, listas, cabeçalhos, imagens, tabelas, ...) sejam formatados e apareçam no documento resultado na mesma ordem em que se encontram no documento original.

Mas, nem sempre é assim. Há situações em que se pretende reordenar estruturalmente o conteúdo dum documento. Pode-se, por exemplo, querer gerar

uma síntese do documento original, ou uma tabela de referências. Este tipo de operações requer uma transformação do documento original. O XSL permite especificá-las.

O XSL adoptou a sintaxe do XML. No entanto, encontra-se ainda em fase de desenvolvimento, podendo ser alterado a qualquer momento. Este facto não assustou, por exemplo a Microsoft que já integrou no seu browser ("Internet Explorer") a possibilidade de processar estilos especificados em XSL.

O XSL resultou da fusão de algumas características de duas outras linguagens de especificação de estilo: do DSSSL (Secção 6.2), e do já descrito CSS.

O XSL combina parte das potencialidades do DSSSL com o CSS e adiciona uma linguagem de programação para ligá-las, o ECMAScript, uma versão standard de Javascript.

Para construir um determinado output da informação guardada num documento XML, um processador XSL irá utilizar uma especificação de estilo para fazer uma travessia ao documento e produzir o output desejado.

Até ao momento, os processadores XSL disponíveis geram apenas outputs em HTML, mas em teoria poderiam gerar qualquer outro formato (à semelhança do DSSSL): RTF, ASCII, PDF, TEX, etc.

Num futuro muito próximo, o XSL (na versão corrente ou com alterações) será suportado directamente nos browsers, de momento isso não acontece.

Em XSL, a associação de uma folha de estilo a um documento XML é feita da mesma maneira que em CSS, através da inclusão da linha:

```
<?xml-stylesheet type="text/xsl" href="curso.xsl"?>
```

D.3.2.1. Anatomia de uma folha de estilo XSL

Eis o conceito mais importante subjacente ao XSL: o XSL não manipula elementos, manipula objectos gráficos ("flow objects"). Mais especificamente, transforma bocados dum documento XML em objectos gráficos.

Surge então a questão: O que são de facto estes objectos gráficos?

Um objecto gráfico é uma unidade no resultado final de uma versão impressa ou colocada na Internet. Por exemplo, quando se formata um parágrafo com um

determinado tipo de letra, ou quando se coloca uma imagem num determinado ponto do écran, não estamos a tratar individualmente nem os caracteres do parágrafo nem os pixeis da imagem.

Um aspecto interessante destes objectos é que podem ser agrupados numa hierarquia de objectos gráficos: uma caixa ("box") pode conter um ou mais parágrafos ("paragraph"). Este aninhamento de objectos começa a parecer familiar, parece mesmo a estrutura dum documento XML. Parece mas não é; um objecto gráfico é a representação pictórica dum componente lógico do documento.

Basicamente, uma folha de estilo XSL é composta por um conjunto de regras de construção. Cada regra tem duas partes, um padrão e um conjunto de acções. Sempre que um elemento do documento original XML corresponder ao padrão especificado numa regra, o correspondente bloco de acções é executado.

Exemplo D-6. Esqueleto de uma folha de estilo XSL

```
<xsl>
  <rule>
    [regra de construção 1]
    [regra de construção 2]
    ...
  </rule>
  <rule>
    [regra de construção 3]
    ...
  </rule>
</xsl>
```

Voltamos ao exemplo do curso e das aulas para demonstrar o que é uma regra de construção.

Exemplo D-7. Regra de construção

```
<rule>
  <target-element type="Aula"/>
```

```
  <DIV font-size="12pt" font-family="sans-serif"
       font-style="italic»
    <children/>
  </DIV>
</rule>
```

As folhas de estilo em XSL podem ser muito simples ou muito complexas. Para mais informações o leitor encontrará no último capítulo dedicado à bibliografia as referências necessárias.

D.4. Extensible Linking Language (XLL) e Extended Pointers (XPointer)

Uma das características que tornou o HTML popular foi a possibilidade de inserir hiperligações numa página que a ligassem a outras páginas. Foi assim que o foi criado o grande manancial de informação que é o WWW.

O XML pretende ir mais além e ultrapassar algumas das limitações do mecanismo de hiperligações do HTML. O XLL tem duas partes: o XLink dedicado à especificação de ligações e o XPointer dedicado ao endereçamento.

O Xlink difere do HTML porque vai tentar ultrapassar as limitações existentes, nomeadamente:

- Ligações multi-direccionais - a possibilidade de uma ligação poder ser atravessada em ambos os sentidos.
- Ligações com destinos múltiplos - dar a possibilidade ao utilizador de escolher um de entre vários destinos.
- Pode operar com um repositório de endereços e toda a gestão a este associada.
- Permite a colocação de ligações "out-of-line- podemos ter um ficheiro à parte onde será colocada a informação das ligações (quais são as suas extremidades); este ficheiro acaba por ser uma lista de pares; este mecanismo é um passo na implementação das ligações bi-direccionais.

O Xpointer vem complementar o Xlink. Normalmente quando temos uma ligação duma página a outra, e se indica ao browser que se quer seguir essa ligação, o browser carrega a nova página integralmente. A ideia por detrás do Xpointer é optimizar esta funcionalidade. Para isso, disponibiliza uma pequena linguagem de query que permite seleccionar qual a parte da nova página que se quer ver. Apresenta-se a seguir um exemplo.

Exemplo D-8. XPointer

```
http://www.di.uminho.pt/~jcr/CURSOS/curso.xml#
       ROOT()CHILD(4,Aula)
```

Este endereço apontaria para a quarta aula no documento estruturado XML de nome curso.xml

---

O Xlink e o Xpointer são bastante complexos e mereceriam um capítulo inteiramente dedicado a eles mas o facto de não serem ainda suportados por nenhuma ferramenta faz deles apenas um objecto de estudo a ter em conta. No entanto, quem quiser ler mais sobre o assunto pode consultar o livro [Bra98].

D.5. NameSpaces in XML (XML Namespace)

Esta é mais uma proposta no sentido de adoptar a filosofia dos objectos para o SGML. A ideia é podermos criar um documento em que uma parte obedeça a um DTD e outras partes obedeçam a outros. Ou seja, podemos criar novas estruturas documentais combinando partes de outras já existentes.

Esta proposta está ainda num estado embrionário e o respectivo suporte a nível de ferramentas de processamento é quase inexistente.

Para terminar apresentamos um exemplo da sua aplicação.

Exemplo D-9. XML NameSpaces

Voltemos ao DTD das receitas. Agora, queremos criar uma nova receita mas queremos colocar para cada ingrediente uma marca aconselhada. Como não previmos nenhum elemento para isso vamos importá-lo do DTD de artigos (descreve os produtos à venda no mercado).

Declaramos a importação de novos elementos:

```
<?xml:namespace name="http://jcr.pt/"
                          href="http://jcr.pt/dtds/artigo.dtd"
                          as="art"?>
```

a partir deste momento todos os elementos do dtd artigo podem ser usados desde que prefixados com "art".

Assim, podíamos escrever:

```
<RECEITAS>
  <TITULO> O Meu Livro de Receitas </TITULO>
  <RECEITA ORIGEM="Portugal»
    <TITULO> Bolo </TITULO>
    <INGREDIENTE> 500g de farinha </INGREDIENTE>
    <art:MARCA>Branca de Neve</art:MARCA>
    <INGREDIENTE> 200g de açucar </INGREDIENTE>
    <art:MARCA>Português</art:MARCA>
    <INGREDIENTE> 300g de manteiga </INGREDIENTE>
    <art:MARCA>Loreto</art:MARCA>
  </RECEITA>
</RECEITAS>
```

D.6. Vector Markup Language

Tem havido um grande esforço para normalizar o formato utilizado para as imagens na Internet. Os normais GIF e JPEG já não satisfazem e a procura de um formato vectorial tem animado empresas e particulares. Decidiu-se que haveria dois formatos, um de alta definição vocacionado para esquemas de maquinaria,

arquitectura e aplicações semelhantes e outro com menos detalhe usado para as imagens que estamos habituados a ver, logotipos, fotografias, esquemas simples.

A guerra pelo formato de alta definição teve poucos concorrentes e ficou rapidamente decidido que seria o CGM.

O outro formato tem sido alvo de bastante polémica. Tudo começou com a proposta da Adobe, o SVG (*"Standard Vector Graphic"*) que logo teve resposta da Microsoft com o VML (*"Vector Markup Language"*). Outras foram também propostas, como a DML (*"Drawing Markup Language"*), mas não motivaram tanto interesse.

Até ao momento, mantêm-se as duas como propostas. De qualquer modo existem implementações das duas em browsers que as suportam.

D.7. Simplified Markup Language - SML

Quando tudo parecia calmo eis que surge nova tempestade.

Quem pensava que o XML já era muito simples e que, provavelmente, seria o formato adoptado por toda a gente enganou-se.

Na passagem do milénio, surge alguém (Robert Quey - [Que99]) que diz que é preciso uma linguagem mais simples.

Eis o que os defensores duma linguagem de anotação simplificada advogam:

- Só deveria usar unicode no formato UTF-8.
- Não deve ter instruções de processamento.
- Não deve ter entidades paramétricas.
- Não deve ter um DTD.
- Não deve ter notações.
- Não deve ter entidades.
- Não deve ter atributos.
- Não deve ter elementos.

No fim, teríamos:

- texto
- comentários (onde o utilizador os quisesse colocar)

É claro que as reacções não demoraram, e Rick Jeliffe [Jel99], rebate esta proposta ponto por ponto. No entanto, será que isto não nos faz lembrar nada?...

Bibliografia

Artigos

[ABNO97] *CAMILA: Formal Software Engineering Supported by Functional Programming*, J.J. Almeida, L.S. Barbosa, F.L. Neves, e J.N. OliveiraEditado por e J. Diaz, Proc. II Conf. Latino Americana de Programacion Funcional (CLaPF97), La Plata, Argentina, October 1997..

[Aho98] *Features of Knowledge Discovery Systems*, Helena Ahonen.

International SGML Users Group (ISUG) Newsletter, 1998.

[BHW99] *Visually Specifying Context*, Anne Bruggemann-Klein, Stefan Hermann, e Derick Wood, March 1999.

[Bray98] *RDF and Metadata*, Tim Bray, Seybold and O'Reilly Publicatons, June 1997.

[Bru94] *Compiler-Construction Tools and Techniques for SGML parsers: Difficulties and Solutions*, Anne Bruggemann-Klein, May 1994.

[Bry98] *Topic Navigation Maps*, Martin Bryan.

International SGML Users Group (ISUG) Newsletter, 1998.

[Cap95] *You Call It Corn, We Call It Syntax-Independent Metadata for Document-Like Objects*, Priscilla Caplan, 1995.

The Public-Access Computer Systems Review 6, 4.

[CKR97] *The Evolution of Web Documents*, Dan Connolly, Rohit Khare, e Adam Rifkin, Seybold and O'Reilly Publicatons, October 1997.

[CCDFPT98] *XML-GL: A Graphical Language for Querying and Reshaping XML Documents*, Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali,

Stefano Paraboschi, e Letizia Tanca, QL'98 - The Query Languages Workshop, December 5, 1998.

[DeR98] *XQuery: A unified syntax for linking and querying general XML documents*, Steven DeRose, QL'98 - The Query Languages Workshop, December 5, 1998.

[DuC97] *Formatting Documents with DSSSL Specifications and Jade: Parts 1, 2, 3*, Bob DuCharme.

<TAG> Newsletter, May, June, July 1997.

[Hee96] *Review of Metadata Formats: Program*, 30, 345-373, Rachel Heery, October 1996.

[Ken96] *Tales from the Front Understanding Structured Documents: <TAG> The SGML Newsletter*, Dianne Kennedy, February 1996.

[Ken97] *An Introduction to DSSSL (ISO/IEC 10179)*, Dianne Kennedy.

<TAG> Newsletter, February 1997.

[Kil99] *SGML & XML content models*, Pekka Kilpelainen99.

Markup Languages: theory and practice, Editado por C. M. Sperberg-McQueen, Editado por B. Tommie Usdin, Spring 1999.

[Knu68] *Semantics of Context Free Languages*, Donald E. Knuth.

Mathematical Systems Theory journal, 1968.

[Knu92] *Literate Programming*, Donald E. Knuth, University of Chicago Press, 1992.

[MA98] *Conceptual Structures and Structured Documents*, Philippe Martin e Laurence Alpay, INRIA - ACACIA project.

[Mar99a] *Construction Rules*, Didier PH Martin, OpenJade Project.

http://www.netfolder.com

[New97] *Document Architectures: the New Hytime*, Steven Newcomb.

International SGML Users Group (ISUG) Newsletter, 1997.

[Omn98] *Defining Microdocument Architecture*, Omnimark Technologies.

http://www.omnimark.com

[Paa95] *Attribute Grammars Paradigms: A High-Level Methodology in Language Implementation*, Jukka Paakki, ACM Computing Serveys, 27, 2, June 1995.

[RAH95] *Algebraic Specification of Documents*, José Carlos Ramalho, José João Almeida, e Pedro Rangel Henriques, TWLT10 - Algebraic Methods in Language Processing - AMiLP95, number 10 in Twente Workshop on Language Technology, Twente University - Holland, Dec. 1995.

[RAH96] *Document Semantics: Two Approaches*, José Carlos Ramalho, José João Almeida, e Pedro Rangel Henriques, SGML96 - Celebrating a Decade of SGML, Boston - USA, Nov. 1996.

[RAH98] *Algebraic specification of documents*, José Carlos Ramalho, José João Almeida, e Pedro Rangel Henriques.

Theoretical Computer Science, 1998.

[RLS98] *XML Query Language (XQL)*, Jonathan Robie, Joe Lapp, e David Schach, QL'98 - The Query Languages Workshop, December 5, 1998.

[RRAH99] *SGML documents: Where does quality go?*, José Carlos Ramalho, Jorge Gustavo Rocha, José João Almeida, e Pedro Rangel Henriques.

Markup Languages: theory and practice, Editado por C. M. Sperberg-McQueen, Editado por B. Tommie Usdin, Winter 1999.

[SAS99] *Designing and Implementing Combinator Languages*, S.Doaitse Swierstra, Pablo Alcocer, e João Saraiva, Advanced Functional Programming

Summer School, Editado por Pedro Henriques, Editado por J.N. Oliveira, Universidade do Minho, 1999.

[Spe98] *A Gentle Introduction to DSSSL*, Dan Speck.

International SGML Users Group (ISUG) Newsletter, 1998.

[Tom98] *Providing flexible access in a query language for XML*, Frank Tompa, QL'98 - The Query Languages Workshop, December 5, 1998.

[TR81] *The Cornell Synthesizer Program: A Syntax-Directed Programming Environment*, Tim Teitelbaum e Thomas Reps, ACM, 24, September 1981.

[Wad99] *A formal semantics of patterns in XSLT*, Philip Wadler, Markup Technologies'99, Philadelphia - USA, Dec. 1999.

[WGMD] *OCLC/NCSA Metadata Workshop Report*, Stuart Weibel, Jean Godby, Eric Miller, e Ron Daniel.

[Wid8] *Querying XML with Lore*, Jennifer Widom, QL'98 - The Query Languages Workshop, December 5, 1998.

Livros

[Bra98] *The XML Companion*, Neil Bradley, Addison-Wesley, 1998.

[DD94] *Making Hypermedia Work: A User's Guide to HyTime*, Steven DeRose e David Durand, Kluwer Academic Publishers, 1994.

[GMS94] *The LaTeX Companion*, Michel Goossens, Frank Mittelbach, e Alexander Samarin, Addison-Wesley, May 1994.

[Gol90] *The SGML Handbook*, Charles Goldfarb, Clarendon Press, 1990.

[Hen92] *Gramáticas de Atributos*, Pedro Rangel Henriques, Tese de Doutoramento, Escola de Engenharia - Departamento de Informática - Universidade do Minho, 1992.

[Her94] *Practical SGML*, Eric Herwijnen, Kluwer Academic Publishers, 1994.

[Lop98] *Gerador de Editores Estruturados de Documentos SGML Definidos por DTDs*, Alda Cristina Reis Santos Lopes, Tese de Mestrado em Informática, Escola de Engenharia - Departamento de Informática - Universidade do Minho, 1998.

[MA96] *Developing SGML DTDs: From Text to Model to Markup*, Eve Maler e Jeanne Andaloussi, Prentice-Hall, 1996.

[Meg98] *Structuring XML Documents*, David Megginson, Prentice-Hall, 1998, 0-13-642299-3.

Charles F. Goldfarb Series on Open Information Management.

[PP92] *The Art of Compiler Design*, Thomas Pittman e James Peters, Prentice-Hall, 1992.

[Ram93] *Um Compilador para o GLiTCH*, José Carlos Leite Ramalho, Tese de Mestrado em Informática, Escola de Engenharia - Departamento de Informática - Universidade do Minho, 1993.

[RT89a] *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Thomas Reps e Tim Teitelbaum, Texts and Momographs in Computer Science., Springer-Verlag, 1989..

[RT89b] *The Synthesizer Generator Reference Manual*, Thomas Reps e Tim Teitelbaum, Texts and Momographs in Computer Science., Springer-Verlag, 1989..

[Sou98] *Semântica de Documentos SGML*, Pedro Rui Marques França Pereira de Sousa, Tese de Mestrado em Informática, Escola de Engenharia - Departamento de Informática - Universidade do Minho, 1998.

[Tar92] *Data Processing in the Unix Environment*, Ramkrishna Tare, McGraw-Hill, 1989.

[SB94] *Guidelines for Electronic Text Encoding and Interchange (TEI P3)*, C.M. Sperberg-McQueen e Lou Burnard , Association for Computers and the

Humanities, Association for Computational Linguistics, Association for Literary and Linguistic Computing, 1994.

[TW95] *The SGML Implementation Guide: A Blueprint for SGML Migration*, Brian Travis e Dale Waldt, Springer, 1995.

[WM99] *DocBook: The Definitive Guide*, Norman Walsh e Leonard Muellner, O'Reilly, 1-56592-580-7, October 1999.

Relatórios

[BA95] *CAMILA: A Reference Manual*, L.S. Barbosa e J.J. Almeida, Technical Report DI-CAM-95:11:2, DI (U.Minho), 1995.

[Fig98] *Publicação na Internet do livro "Memórias de José Inácio Peixoto dos Santos"*, Nuno Figueiredo, Arquivo Distrital de Braga / Universidade do Minho, Fevereiro de 1998.

[HAR99] *DAVID - Manipulação Algébrica de Documentos: Projecto JNICT - 2479/96*, Pedro Rangel Henriques, José João Almeida, e José Carlos Ramalho, Departamento de Informática - Universidade do Minho, 1996-1999.

[LC99] *Recuperação de Edições Esgotadas: Inventário das Cartas do Cabido da Sé de Braga*, Armando Lemos e José Henrique Cerqueira, Arquivo Distrital de Braga / Universidade do Minho, Fevereiro de 1999.

[LO99] *Publicação de Edições Esgotadas do Arquivo Distrital de Braga na Internet: Inventário das Visitas e Devassas do Cabido da Sé de Braga*, Carla Lopes e Clara Oliveira, Arquivo Distrital de Braga / Universidade do Minho, Fevereiro de 1999.

[RG98] *Publicação na Internet do Livro: "Ensaio Sobre as Minas de Joze Anastacio da Cunha*, Salomé Ribeiro e Sónia Gaspar, Arquivo Distrital de Braga / Universidade do Minho, Julho de 1998.

[Sal97] *SGML/XML tools and comparision: Design and development of SGML conversions*, Carlos Salgado, FAST - Faculdade de Wuerzburg - Alemanha /

Universidade do Minho, 1997.

Publicações Electrónicas

[ANTLR] "*ANTLR- (http://www.ANTLR.org)*

[ARCSGML] "*ARCSGML- (ftp://ftp.ifi.uio.no/pub/SGML/ARC-SGML)*

[ASP] "*ASP-SGML: Jos Warmer's Amsterdam SGML Parser- (ftp://ftp.ifi.uio.no/pub/SGML/ASP-SGML)*

[Bray98] "*When is an attribute an attribute?*- Tim Bray; *(http://www.oasis-open.org/cover/brayAttr980409.html)*Robin Cover; OASIS; 1998/04/09

[CMAlgebra] "*Content Model Algebra- (http://www.omnimark.com/white/content/)*Omnimark White Papers; Omnimark Technologies; 1998

[css1] "*Cascading Style Sheets Level 1 (CSS1)*- World Wide Web Consortium Recommendation 17-December-1996; *(http://www.w3.org/TR/REC-CSS1-961217)*

[css2] "*Cascading Style Sheets Level 2 (CSS2)*- World Wide Web Consortium Recommendation 12-May-1998, *(http://www.w3.org/TR/1998/REC-CSS2-19980512/)*

[DuC98] "*DBMS support of SGML*- Bob DuCharme; *(http://www.oasis-open.org/cover/ducharmeVendors9808.html)*Robin Cover; OASIS; 1998/08

[DCD] "*Document Content Definition- (http://www.w3c.org/TR/1998/NOTE-dcd-19980731.html)*World Wide Web Consortium Note; World Wide Web Consortium; 1998.07.31

[DLMC] "*Digital Library Miguel de Cervantes- (http://www.w3c.org/TR/1998/NOTE-dcd-19980731.html)*Universidade de Alicante // Banco Santander Central Hispano; 1999-2000

[DocBook] "*The DocBook DTD- (http://www.oasis-open.org/docbook/)*OASIS; 1998

[DOM] "*Document Object Model (DOM)- (http://www.w3c.org/DOM/)*

[Cla96] "*DSSSL - Document Style and Semantics Specification Language*- Editado por James Clark; *(http://www.jclark.com/dsssl)*; 1996

[Mul98] *DSSSL Documentation Project (http://www.mulberrytech.com/dsssl/dsssldoc/index.html)*Mulberry Technologies; 1998

[GEIRA] "*Gestão da Informação da área fronteiriça norte*- Universidade do Minho; *(http://www.geira.pt/)*; 1996 - 1999

[Ger96] "*An Introduction to DSSSL*- Daniel Germán; *(http://www.jclark.com/dsssl)*; 1996

[Hol99] "*When to use attributes as opposed to elements*- G. Ken Holman; *(http://www.oasis-open.org/cover/holmanElementsAttrs.html)*Robin Cover; OASIS; 1999/01/11

[html4.0] "*HyperText Markup Language Version 4.0 (HTML)*- *(http://www.w3.org/TR/1998/REC-html40-19980424/)*World Wide Web Consortium Recommendation 24-Apr-1998;

[jade] "*Jade*- James Clark; *(http://www.jclark.com/jade/)*; 1996-99

[Jel99] "*Goldilocks and SML*- Rick Jeliffe; *(http://www.xml.com/pub/1999/12/sml/goldilocks.html)*; Dec. 15 1999

[Kim97] "*Designing a DTD: Elements or attributes?*- W. Eliot Kimber; *(http://www.oasis-open.org/cover/attrKimber9711.html)*Robin Cover; OASIS; 1997/11/18

[Meg98b] "*PSGML - DSSSL*- David Megginson; *(http://www.oasis-open.org/cover/megg-psgml-dsssl-el.txt)*; 1998.02.23

[OpenJade] "*OpenJade- (http://www.netfolder.com/DSSSL/index.html)*

[POSTSCRIPT] "*Postscript- (http://www.adobe.com)*

[Pre97] "*Introduction to DSSSL*- Paul Prescod; *(http://www.jclark.com/dsssl)*; 1997

[Que99] "*SML: Simplifying XML*- Robert Quey; *(http://www.xml.com/pub/1999/12/sml/index.html)*; Nov. 24 1999

[SGML.Decl] "*Understanding The SGML Declaration*- *(http://www.omnimark.com/white/dec/)*Omnimark White Papers; Omnimark Technologies; 1998

[SGMLS] "*SGMLS*- James Clark; *(http://www.jclark.com/jade/)*

[SP] "*SP - SGML Parser*- James Clark; *(http://www.jclark.com)*; 1995-99

[Spy97] "*Formal Public Identifier Roadtrip- (http://www.cm.spyglass.com/doc/fpi.html)*Cambridge Spyglass Technical Reference; Spyglass Inc.; 1997

[Sta99] "*PSGML- (http://www.lysator.liu.se/projects/about-psgml.html)*; 1999.10.14

[Tho97] "*SGML Groves: A Partial Illustrated Example- (http://www.cogsci.ed.ac.uk/~ht/grove.html)*Henry Thompson; 1997

[Unicode] "*Unicode - (http://www.unicode.org)*

[Wal2000] "*Modular Docbook DSSSL Stylesheets- (http://nwalsh.com)*

[WIDL] "*Web Interface Definition Language- (http://www.w3c.org/TR/1998/NOTE-widl-19970922.html)*World Wide Web Consortium Note; World Wide Web Consortium; 1997.09.22

[xlink] "*Extensible Linking Language (XLink)- (http://www.w3.org/TR/1998/WD-xlink-19980303)*World Wide Web Consortium Working Draft 3-March-1998;

[XML] "*eXtended Markup Language (XML)- (http://www.xml.com)*

[xml1.0] "*Extensible Markup Language (XML) Version 1.0 - (http://www.w3.org/TR/1998/REC-xml-19980210.html)*World Wide Web Consortium Recommendation 10-February-1998;

[xpointer] "*XML Pointer Language (XPointer)- (http://www.w3.org/TR/1998/WD-xptr-19980303)*World Wide Web Consortium Working Draft 3-March-1998;

[xsl] "*Extensible Stylesheet Language (XSL) Version 1.0- (http://www.w3.org/TR/1998/WD-xsl-19980818)*World Wide Web Consortium Working Draft 18-August-1998;

[xslt] "*XSL Transformations (XSLT) - Version 1.0- (http://www.w3.org/TR/1999/REC-xslt-19991116.html)*World Wide Web Consortium Recommendation 16-November-1999;

[YASP] "*YASP: Pierre Richard's Yorktown Advanced SGML Parser (or: 'Yet Another SGML Parser')- (ftp://ftp.edf.fr/pub/SGML/YASP)*

[YAO] *"YAO (Yuan-Ze–Almaden–Oslo project) Parser Materials-
(ftp://hki.wsoy.fi/pub/yao-u.tar.gz)*