Engenharia reversa de HTML usando tecnologia XML

José João Almeida and Alberto M. Simões

Projecto Natura
Departamento de Informática
Universidade do Minho
{jj@|albie@alfarrabio.}di.uminho.pt
http://natura.di.uminho.pt

Resumo O proliferar de ferramentas criadores de HTML e o uso de HTML guiado pelo aspecto, tem vindo a arruinar o seu lado conceptual. Este problema foi reconhecido e deu origem a vários formatos ou tecnologias com o objectivo de separar o aspecto do conceito.
No entanto a realidade actual mostra uma enorme quantidade de páginas HTML com péssima leitura conceptual e estrutural, invalidando uma série de usos possíveis da informação nelas contida.
Nesta comunicação apresenta-se um trabalho (em fase inicial) que pretende fazer engenharia reversa de HTML para permitir aumentar a sua acessibilidade, a fim de ser usada num *browser* para invisuais.

1 Introdução

O uso abusivo de HTML para estruturar graficamente a informação tem tornado grande parte das páginas ilegível mesmo para pessoas sem deficiência visual. Embora o *World Wide Web Consortium* e outras entidades tenham apresentado várias tecnologias para separar o lado conceptual do lado gráfico das páginas de Internet (HTML[4] com CSS[3] ou XML[1] com XSL[2]), o seu uso é ainda bastante reduzido.

Em particular, os portais, os jornais e páginas de informação são das que mais saturaram o aspecto visual descuidando completamente o lado conceptual. No entanto, estas páginas são as que mais interessam ao público em geral (e em particular aos invisuais).

O nosso objectivo principal é construir um conjunto de ferramentas para simplificar o código HTML.

Em vez de construir um ferramenta simplificadora de HTML, optou-se por alargar o domínio de uso de uma ferramenta de processamento de XML (o XML::DT), levando a que o processamento estrutural de XML que ela permite possa ser aplicável ao HTML. Deste modo a criação de ferramentas de simplificação e processamento de HTML, passa a ser programar XML::DT, permitindo uma escrita compacta e usufruindo de uma experiência e de uma cultura já existente.

Neste documento, para além de exemplos ligados a acessibilidade, serão ainda apresentados alguns exemplos de manipulação geral de HTML, sendo nossa intenção demonstrar que é simples a escrita destes processadores.

As ferramentas aqui apresentadas foram construídas usando um módulo Perl de processamento de documentos XML denominado de XML::DT[5]. Este módulo funciona sobre a biblioteca `libxml2`, equipada com fuzzy-parsing, pelo que é capaz de processar HTML com erros.

2 Introdução ao `XML::DT`

Este documento vai introduzir o uso do módulo apresentando exemplos progressivamente mais complicados. Todos estes exemplos serão de processamento e limpeza de documentos HTML. Embora a compreensão total de determinados exemplos obrigue a conhecimentos da linguagem Perl, é perfeitamente possível entender a mensagem deste artigo através duma compreensão parcial.

Por uma questão de simplificação apenas usaremos as seguintes funções do módulo `XML::DT`:

- `dt` – (down-translate) dado um ficheiro (XML ou HTML) e um processador, faz a travessia e processamento estrutural.
- `dturl` – idem a partir do url dum documento
- `toxml` – gera XML.

e das variáveis globais:

- `$q` – nome da etiqueta do elemento actualmente em processamento
- `$c` – conteúdo do elemento actualmente em processamento
- `%v` – array associativo dos atributos do elemento actualmente em processamento

2.1 Remoção de etiquetas

No primeiro exemplo faz-se a remoção das etiquetas `script` e a substituição das etiquetas de imagem pelo seu atributo `alt`. Este atributo, frequentemente ignorado, é fulcral para permitir a legibilidade a invisuais.

```
1 #!/usr/bin/perl
2  use XML::DT;

3  %h=( -html      => 1,
4       -outputenc => "ISO-8859-1",
5       img        => sub{ $v{alt} || "" },
6       script     => sub{ "" } );

7  print dt(shift,%h);
```

A primeira linha, comum a todas as *scripts* (de qualquer linguagem), indica qual o interpretador da linguagem a ser usado. No nosso caso, o Perl. A linha seguinte importa o módulo que vamos usar[1].

As quatro linhas seguintes configuram o processador do documento XML. A primeira opção, denominada por `-html` instrui o módulo de que deve utilizar o fuzzy-parsing para ler documentos HTML mal formados. De seguida indicamos qual a codificação a utilizar na apresentação dos resultados. Segue-se a definição de duas funções para processar as etiquetas em questão (`img` e `script`. No caso desta última, associamos-lhe uma função que retorna a *string* vazia (pelo que esta etiqueta e seu conteúdo é removido). No caso da etiqueta `img`, retornamos o seu atributo `alt`, acessível a partir do *array* associativo `v` ou, caso não exista, a *string* vazia.

A última linha imprime o resultado de processar o ficheiro passado como argumento.

Como se observa, o facto de se poder definir transformações locais (associadas a um contexto expecífico) simplifica bastante a acção de transformar HTML. Às etiquetas que não associamos uma função de processamento é associada automaticamente a função identidade.

2.2 Remoção de atributos e etiquetas

Uma das coisas que tornam o HTML complexo é a utilização de uma série infindável de atributos visuais.

No exemplo seguinte vão ser removidos atributos que constem de uma lista de atributos *ignoráveis*.

```
1  my @ignore = qw/color width height/;

2  %h = ( -html => 1,
3         -outputenc => 'ISO-8859-1',
4         -default => sub{
5            for (@ignore) { delete($v{$_}) }
6            toxml
7         });

8  print dt(shift,%h);
```

Este exemplo mostra o uso da regra `-default` que é executada para todas as etiquetas que não têm função de processamento especificada. Mostra também a função `toxml` que de acordo com o nome da etiqueta em causa `$q`, do *array* associativo de atributos `%v` e do conteúdo da etiqueta `$c`, reconstrói o texto XML. Note-se que depois de apagar determinados atributos estes deixam de existir e portanto não aparecem na reconstrução do texto.

Embora funcione, este exemplo é bastante simplicista. Na verdade, o que vamos querer é remover determinadas etiquetas (como as `script`), determinados atributos (como o `color`) e alguns atributos para determinados elementos.

[1] nos exemplos seguintes omitiremos essas linhas para simplificar.

```
1  my %ignore_tag = (script => 1);
2  my %ignore_atts => ( -default => ['color','width','height'],
3                       a => ['onmouseover', 'onmouseclick'] );

4  %h = ( -html => 1,
5         -outputenc => 'ISO-8859-1',
6         -default => sub{
7            my @atts_to_remove = ();

8            return "" if $ignore_tag{$q};

9            if (exists($ignore_atts{$q}))
10            { @atts_to_remove = @{$ignore_atts{$q} }
11           else
12            { @atts_to_remove = @{$ignore_atts{-default}} }

13           for (@atts_to_remove) { delete($v{$_}) }
14           toxml
15        });

16 print dt(shift,%h);
```

Descrição do programa:

linha 1: definimos um *array* associativo em que as chaves são as etiquetas a remover — usar um *array* associativo em vez de um simples *array* torna a consulta mais rápida;

linha 2-3: definimos também um *array* que associa a cada etiqueta uma lista de atributos a remover. A chave `-default` corresponde aos atributos a remover por omissão;

linha 7: na primeira linha da função de processamento de todas as etiquetas definimos um *array* local dos atributos a remover;

linha 8: se a etiqueta em causa (acessível na variável `$q`) é para remover, retornar a *string* vazia;

linha 9-12: verificar se existe uma regra específica de atributos a remover para a etiqueta em causa. Se não existir, usar a regra por omissão;

linha 13-14: semelhante ao exemplo anterior, apaga os atributos em causa e retorna o texto XML correspondente;

2.3 Tratamento de `frames` e `iframes`

A tecnologia de *frames* é das que levantam complicações em relação à forma como devem ser tratadas para que um invisual consiga perceber minimamente o conteúdo do *site* em questão.

Nesta subsecção vamos apresentar um exemplo (simplicista) que se baseia em:

– todas as *frames* vão ser concatenadas da esquerda para a direita e de cima para baixo;

– as *iframes* vão ser incluídas directamente no local onde aparecem no documento;
– estas inclusões devem ser processadas recursivamente.

```
1  use LWP::Simple;

2  %h = ( -html => 1,
3         -outputenc => 'ISO-8859-1',
4         frameset => sub { "$c" },
5         frame  => sub { dturl($v{src}, %h) },
6         iframe => sub { dturl($v{src}, %h) },
7         noframes => sub { "" } );

8  print dt(shift,%h);
```

Descrição do programa:

linha 1: precisamos de um novo módulo Perl para fazer download de páginas;
linha 4: as etiquetas `frameset` desaparecem, mantendo o seu conteúdo;
linha 5,6: para cada um dos tipos de `frames`, chamar uma função que processe o URL do atributo `src`;
linha 7: por fim, remover a etiqueta `noframes`, desnecessária para o exemplo em questão, visto que iria introduzir redundância ao documento (e também porque nunca é usada da forma correcta);

2.4 Extracção de sub-tabelas

Em muitas páginas que são constituídas por tabelas, há utilidade em extrair partes, por exemplo em extrair o corpo central, ou uma sub-tabela que seja o índice.

No próximo exemplo apresenta-se um programa que quando usado sem argumentos constrói uma página HTML com a lista de todas as tabelas existentes (e respectivos identificadores); quando invocado com um um identificador de tabela, constrói um página HTML contendo apenas essa tabela.

Os identificadores de tabela usados estão a ser constituídos por um numeração composta de acordo com a hierarquia das tabelas. Exemplo: 1.1.3 – 3ª tabela contida na 1ª tabela da 1ª tabela. Este tipo de identificador é mais estável às mudanças locais.

```
1  my $file= shift or die("usage tabela file.html [tableid]\n");
2  my $tn=shift;

3  %h1=(-html => 1,
4       -outputenc => 'ISO-8859-1',
5       -begin => sub{ print "<html><body>" },
6       -end   => sub{ print "</body></html>" },
7       table  => sub{ print "\n<h1>Tab. ",ntb()."</h1>\n",toxml();
8                      toxml() });
```

```
9  %h2=(-html => 1,
10      -outputenc => 'ISO-8859-1',
11      table => sub{
12        if($tn eq ntb()){
13            print "<html><boby>",toxml(),"</body></html>"; exit; }
14        else {toxml()} });

15 if($tn){ dt($file,%h2);}  ## extrai a tabela $tn
16 else   { dt($file,%h1);}  ## constrói uma pág. com a lista das tabelas

17 sub ntb{
18    my $l=$dtcontextcount{table};
19    for (0..$l-1){$na[$_] ||= 1 ;  }
20    @na = (@na[0..$l-2], $na[$l-1]+1);
21    join(".", (@na[(0..$l-2)],$na[$l-1]-1));
22 }
```

Descrição do programa:

linha 2-7: Definição do processador para a situação em que não é dado o identificador de tabela pretendido.
linha 4: Código executado no ínicio do processamento.
linha 5: Código executado no fim do processamento.
linha 6-7: função de processamento de tabela: imprimir um cabeçalho com o identificador da tabela seguido da tabela (efeito lateral) e devolve o texto da tabela.
linha 8-13: Processador para a situação em que é dado o identificador de tabela a extrair.
linha 10-13: função de processamento de tabela: se o identificador de tabela é o pretendido, imprime a tabela (efeito lateral) e sai.
linha 14-15: Conforme há ou não identificador de tabela, invoca o processador pretendido.
linha 16-23: Ignorar! (Define a função que calcula o identificar da tabela corrente.

Esta script pode ser usada do seguinte modo

```
1  tabelas publico.html > listaDeTab.html
```

para a construção da lista das tabelas encontradas, permitindo observar qual ou quais as que mais interessem ao fim em vista. Para extrair uma tabela estecífica, usaremos:

```
1  tabelas publico.html 1.3 > tabela.html
```

3 `XML::DT` com XPath

Depois de muito se ter usado o módulo `XML::DT` verificou-se que algum do código escrito em muitos dos processadores construídos podiam ser simplificados com a utilização do XPath. Com esse objectivo foi desenvolvida uma função de processamento específica para aceitar (um subconjunto de) predicados XPath.

Esta nova função (`pathdt`) faz uma análise dos predicados em questão e gera o código necessário para que a função genérica do `XML::DT` realize as transformações especificadas.

3.1 Criação de índices

À semelhança do que acontece com quem vê, a leitura de uma página por parte de um invisual não é normalmente uma leitura de sequencial. A página deve ser trabalhada e seccionada, associando um título a cada uma destas secções. Estes títulos é que serão lidos sequencialmente, permitindo ao ouvinte escolher que secções quer ler.

O exemplo sequinte tem como objectivo processar um documento HTML e colocar o seu índice no seu início;

```
1  my @index = ();
2  my $i=0;
3  %h = ( -html => 1,
4         -outputenc => 'ISO-8859-1',
5         'h1|h2|h3' => sub{
6             $index[$i] = toxml;
7             $c = toxml("a",{name=>"a$i"},$c);
8             $i++;
9             toxml; },
10        body => sub{
11            my $index = "";
12            my $j = 0;
13            for (@index) {
14                $index.="<a href='a$j'>$_</a>"; $j++
15            }
16            $c = "$index $c";
17            toxml;
18        }
19 );

20 print pathdt(shift, %h);
```

Embora este exemplo seja um pouco *naïve*, demonstra a utilidade do XPath para determinadas situações, bem como a importância de ferramentas deste género.

Descrição do programa:

linha 1,2: inicializar uma lista com as entradas do índice, e inicializar o contador de entradas;
linha 5: utilizar um predicado XPath para associar uma função às etiquetas `h1`, `h2` e `h3`;
linha 6: guardar no índice o título (e respectiva etiqueta de *heading*) na lista de entradas do índice;
linha 7-9 : colocar uma âncora de destino no título;
linha 11-16: construir o índice e colocá-lo no início do documento;

3.2 Secções por `font`

É habitual nos tempos que correm, determinadas ferramentas geradoras de HTML (como o *Microsoft Word*) produzirem HTML em que os títulos não são nada mais do que uma etiqueta `font` a definir determinado tamanho.

Podemos tentar fazer engenharia reversa com o seguinte código:

```
%h = ( -html => 1,
       -outputenc => 'ISO-8859-1',
       "//font[@size&gt;'4']" => sub { $q = 'h1'; toxml },
       "//font[@size='4']"    => sub { $q = 'h2'; toxml },
       "//font[@size='3']"    => sub { $q = 'h3'; toxml },
       "//font[@size='2']"    => sub { $c               },
       "//font[@size='1']"    => sub { $q = 'small'; toxml },
);

print dt(shift, %h);
```

Este código, fortemente baseado em predicados XPath, associa funções diferentes às etiquetas `font` dependendo do valor do seu atributo `size` (que segundo a especificação, varia entra 1 e 7). Embora em Perl não fosse necessário utilizar a entidade ">" para representar o sinal, optou-se por manter coerência com a especificação do XPath.

4 `XML::DT` Tipado

O `XML::DT` pode associar tipos elementos XML, permitindo que seja automaticamente construída uma representação perl de ficheiro XML (ou de partes dele. Considere-se o seguinte documento XML:

```
<a>
  <b>1</b>
  <b>2</b>
  <b>3</b>
  <b>4</b>
</a>
```

Por omissão a função de processamento de cada elemento recebe uma string com o resultado de processar os filhos. Se associarmos o tipo `SEQ` ao elemento "a", a função que o processa passa a receber a lista `[1,2,3,4]`[2] como argumento (em `$c`).

Os tipos suportados de base pelo `XML::DT` são:

STR é o tipo por omissão e concatena todos os valores retornados pelos seu sub-elementos, o que implica que os seus sub-elementos devem retornar uma string;

SEQ constrói uma lista (array) com o conteúdo dos sub-elementos. Isto significa que os atributos são ignorados (ou devem ser processados no sub-elemento). Retorna a referência para a lista.

SEQH constrói uma lista de arrays associativos com todos os seus sub-elementos. Cada elemento da lista é um array associativo que associa:

[2] Em perl esta lista é uma referência.

-q ao nome do elemento;
-c ao seu conteúdo;
at1 ao valor do atributo `at1` (para cada atributo existente);

MAP cria um array associativo com os sub-elementos; as chaves são os nomes dos sub-elementos e os valores os seus conteúdos. Os atributos são ignorados (devem ser processados nos sub-elementos);

MULTIMAP cria um array associativo de listas; as chaves são os nomes dos sub-elementos e os valores são listas com os conteúdos dos vários elementos com o mesmo nome. Os atributos são ignorados.

MMAPON é escrito como `MMAPON(lista-de-elementos)`, e cria um array associativo com os sub-elementos. As chaves são os nomes dos sub-elementos e os valores os seus conteúdos. Os atributos são ignorados. Para todos os elementos contidos na lista de elementos, é criado uma lista com os seus conteúdos;

XML retorna uma referência para um array associativo com:
-q ao nome do elemento;
-c ao seu conteúdo;
at1 ao valor do atributo `at1` (para cada atributo existente);

ZERO não processa os sub-elementos. Retorna ;

4.1 HTML com design por Tabelas

Nas páginas HTML aparece frequentemento o uso de tabelas para compor gráficamente pedaços HTML. Esse sistema complica o seu uso em ambientes não gráficos.

No próximo exemplo cada tabela está a ser analisada e transformado de acordo como as suas dimensões, seguindo a eurística que a seguir se descreve:

- tabelas 1×1 — dão origem a um título (`h1`);
- tabelas $1 \times n$ — dão origem a uma lista (`ul`);
- tabelas $2 \times n$ — dão origem a uma lista descritiva (`dl`) se a opção `mapping_as_dl` tiver sido selecionada;
- tabelas $n \times n$ — são mantidas como tabelas;

Para que o cálculo das dimensões da tabela fosse fácil, estamos a associar tipos a algumas etiquetas:

- cada `table` vê os seus filhos como uma sequência (de `tr`)
- cada `tr` vê os seus filhos como uma sequência (de `td`)

Deste modo o tipo associado é estruturado e pode ser facilmente usado para determinar o número de linhas e colunas.

```
1 use CGI qw(:all) ;

2 our ($mapping_as_dl);

3 my $filename = shift;
```

```
4  %h=( -html => 1,
5       -outputenc => 'ISO-8859-1',
6       -type  => { table => "SEQH",
7                   tr    => "SEQH",
8                 },
9       tr     => sub{ $c},
10      td     => sub{ $c},
11      table  => sub{
12         my ($li,$co)=dimTabela($c);
13         print STDERR "Debug Tabela($li,$co)";
14         if($li == 1 && $co == 1){ h1($c->[0]{-c}[0]{-c}) }
15         elsif($co == 1)         { ul(li([map {$_->{-c}[0]{-c}} @$c])) }
16         elsif($li == 1)         { ol(li([map {$_->{-c}} @{$c->[0]{-c}}])) }
17         else                    {
18           if($co==2 && $mapping_as_dl){
19               dl(map {CGI::dt($_->{-c}[0]{-c}).dd($_->{-c}[1]{-c}) } @$c) }
20           else { toxml()}}
21      },
22    );

23 print dt($filename,%h);

24 sub dimTabela{
25   my $t=shift;
26   my $nrows = @$t;
27   my $ncolumns = (sort map { scalar(@{$_->{-c}}) } @$t)[-1];
28   ($nrows,$ncolumns);
29 }
```

Descrição do programa:

linha 1: Está a ser usado o módulo perl para gerar HTML (permitindo o uso de funções com nome igual às várias etiquetas HTML como h1, ul, li, dl, etc)
linha 2: Declara-se uma opção de linha de comando.
linha 6 a 7: definição de tipos e modo de processamento de table e tr. Esta definição leva a que o `$c` correspondente seja uma lista de listas que é processada na entrada `table` e na função `dimTabela`.
linha 11 a 21: Processamento de tabelas...
linha 12: determinação das dimensões da tabela
linha 14: ... tabela 1×1 – faz um cabeçalho
linha 15: ... tabela $1 \times n$ – faz uma unorded list
linha 16: ... tabela $n \times 1$ – faz uma orded list
linha 18: ... tabela $2 \times n$ – faz uma description list
linha 19: ... tabela $n \times n$
linha 20: Apesar de `$c` ser uma lista de listas, a função `toxml` sabe reconstituir o XML (HTML) original.
linha 24 a 29: Função determinadora das dimensões da tabela. Ignorar os detalhes (está a ser usado perl avançado...)

Considere-se o seguinte documento HTML:

```
<body>
  <table width="100%" bgcolor="red" border="1">
    <tr> <td> Um estranho título (table 1 x 1) </td> </tr>
  </table>
  <p>Seguidamente uma tabela normal (2 x 3)</p>
  <table border="1">
    <tr> <td> Tomates </td> <td> Vermelhos </td> </tr>
    <tr> <td> Couves  </td> <td> Verdes    </td> </tr>
    <tr> <td> Pencas  </td> <td> Amarelas  </td> </tr>
  </table>
  <p>Seguidamente uma tabela com uma coluna</p>
  <table border="1">
    <tr> <td> Sopa            </td> </tr>
    <tr> <td> ovos estrelados </td> </tr>
    <tr> <td> Pudim           </td> </tr>
  </table>
</body>
```

Na figura 1 mostra-se o resultado de visualizar a página depois de processada (com a opção `-mapping_as_dl`) e apresenta-se também o aspecto inicial.

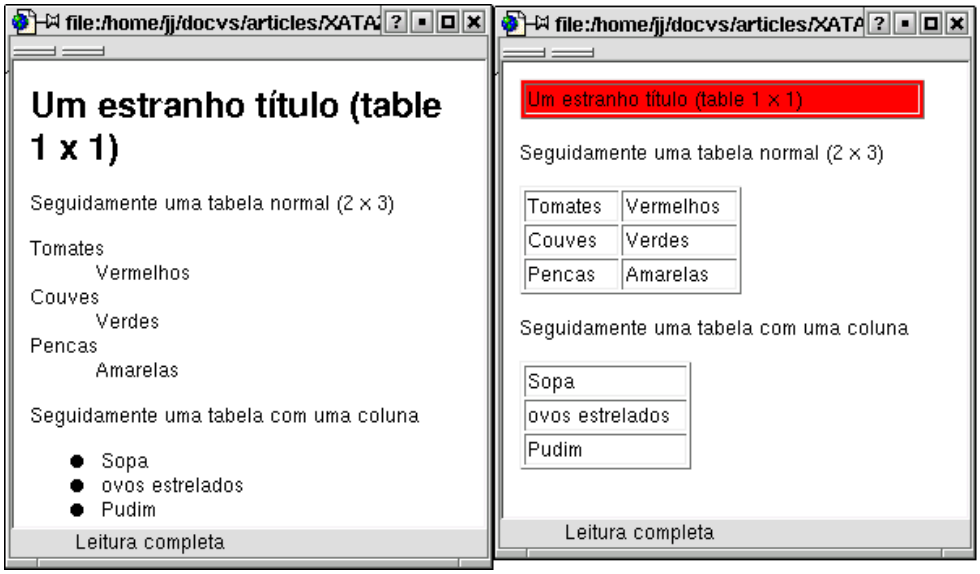

Figura 1. Exemplo de tabelas

5 Conclusões e trabalho futuro

Os exemplos aqui apresentados são propositadamente pequenos e simplicistas mas acreditamos que demonstram as potencialidades da abordagem. Todos estes exemplos terão de cooperar numa mesma ferramenta para que seja possível processar páginas obtendo resultados realmente interessantes.

O objectivo desta ferramenta é ser incluída num serviço de *Proxy* para *browsers* de invisuais que torne uma página HTML complicada numa mais simples de ler por um sintetizador de voz, e para situações em que a acessibilidade seja crítica.

Está a decorrer também um conjunto de testes que usam esta abordagem num leque mais vasto de problemas ligados a processamento de HTML, incluindo web-mining.

Referências

1. *eXtended Markup Language (XML) version 1.0 recommendation.* World Wide Web Consortium, 10 February 1998. http://www.w3.org/TR/1998/REC-xml-19980210.html/.
2. *Extensible Stylesheet Language (XSL), Version 1.0.* World Wide Web Consortium, 15 October 2001. http://www.w3.org/TR/xsl/.
3. *Cascading Style Sheets, level 1.* World Wide Web Consortium, 17 January 1999. http://www.w3.org/TR/REC-CSS1.
4. *HyperText Markup Language Version 4.0 (HTML) recommendation.* World Wide Web Consortium, 24 April 1998. http://www.w3.org/TR/1998/REC-html40-19980424.
5. J.J. Almeida and José Carlos Ramalho. XML::DT a perl down-translation module. In *XML-Europe'99, Granada - Espanha*, May 1999.