XML::DT - a Perl down translation module

José João Dias de Almeida *   José Carlos Ramalho †

April 23, 1999

Abstract

In this paper we present a Perl module, called XML::DT, that can be used to translate and transform XML documents.

A programmer always looks for the simplest tool to do a certain task, development and maintenance will be easier. That is the main idea behind the module we are presenting: a simple tool that will enable the user to speed up his work and that will help him to maintain it.

XML::DT includes some down translation features that are common to other SGML/XML processors available on the market like *omnimark* [**?**] or *balise* [**?**], and some other features to deal with input and output of Unicode character sets.

The idea was to adopt familiar concepts together with a familiar syntax to SGML/XML programmers but shaped to the usual Perl notation.

1 Introduction

There are many tools to process SGML/XML but in the remaining of this document when talking about the history of tool development we will only consider shareware and free tools.

Perl is an unquestionable tool when we are talking about text processing in general or being more specific, structured documents processing. In recent times the interest for this has increased a lot. For some years we had David Megginson's Perl library and module to process SGML documents [**?**] - in fact we could use these modules to create processors that would process the output of SGMLS and NSGMLS parsers [8]. SGML was difficult to process so the interest in tool development was low. But then, XML emerged and XML is a lot easier to process: DTD syntax has changed, annoying things like the ampersand operator were left out, and you can even create and process the document without a DTD. The scenery was set for tool development.

New tools start appearing and other SGML tools suffered some changes so they can process XML aswell. However XML tools developed from scratch are simpler and easier to use.

* (`jj@di.uminho.pt`)
† (`jcr@di.uminho.pt`)

One of the first was the XML Language Toolkit from Henry Thompson [?]. It provided a set of small tolls that could be combined to process XML documents.

Concerning Perl universe everything started with the work from Larry Wall and Clark Cooper. They developed a Perl module that exports the necessary functionality to XML parsing: XML::Parser [?].

XML::Parser is built upon a C library, expat, that is very fast and robust. Expat was authored by James Clark [?], a highly respected developer and consultant in the SGML/XML community.

Since then, many Perl modules were developed upon XML::Parser. XML::DT belongs to this family.

XML::DT is built on XML::Parser. The main idea was to provide developers with a down-translation skeleton and programming shortcuts. *dt* (down-translation) is XML::DT main function, it takes an XML document and a processor specification as arguments and returns the translated document.

In the next sections we will go deeper explaining how XML::DT was built and we will present several examples of use with growing complexity.

To fully understand the remainder of this document some familiarity with Perl is needed even though we have tried to comment everything.

2 A flavor of XML::DT use

The basic function of the module is called `dt`. The parameters of `dt` are the document filename and a processor specification which is a mapping of elements to anonymous functions (perl sub). The mapping may include a "-default" function defining a default translation function (to be used for elements out of the mapping domain).

In order to make the code simpler to write, every function is associated with an element name and can use the global variables `$c` – for element content, `$v{atrName}` – for attribute value, and `$q` – for the element generic identifier.

One important feature of Perl [?], expat and XML::Parser is that *they are all Unicode-aware* ; that is, they can read encoding declarations and perform the necessary conversions into Unicode [?], a system for *the interchange, processing, and display of the written texts of the diverse languages of the modern world* . Thus a single XML document written in Perl can now contain Greek, Hebrew, Chinese and Russian in their proper scripts.

Unfortunately many other tools and environment are not Unicode aware. In XML::DT a output encoding option ("`-outputenc`") is possible, but should be used just in special cases.

In a similar way, "`-inputenc`" (implemented in XML::Parser module) makes it possible to force a input encoding type. Whenever possible, the user should define the input encoding in the XML file:

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

In the next subsections we present a series of examples with growing complexity. In this examples we will try to illustrate the implemented features of our module together with its potential.

2.1 Extracting meta information from a paper

Let's consider the following xml example of a simplified paper to be submitted to a workshop:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<article>
 <title>The XML Down Translator</title>
 <author>J. João Almeida</author>
 <author>J. Carlos Ramalho</author>
 <keyword>XML</keyword>
 <keyword>language processing</keyword>
 <keyword>perl</keyword>
 <abstract>
  Once upon a time ...
 </abstract>
</article>
```

The following perl program (using XML::DT) can be used to extract some meta-information in order to build a bibliographic reference in HTML:

```
 1  #!/usr/bin/perl
 2  use XML::DT ;
 3  my $filename = shift;

 4  %handler=(
 5       '-outputenc' => 'ISO-8859-1',
 6       '-default'   => sub{""},
 7       'title'      => sub{"<b>$c</b>"},
 8       'author'     => sub{"  <i>$c</i>"},
 9       'article'    => sub{"$c<br>"}
10     );
11  print dt($filename,%handler);
```

The functions defined in lines 7 to 9 just put HTML tags around element content (`$c`). Many problems can be solved with functions so simple as these ones.

In line 6 we have defined a general function stating that by default, each element content should be suppressed.

Line 5, we force the output in ISOlatin1. This emergency option was used to process our names in an environment that is not totally Unicode aware. Whenever possible this situation should be avoid.

In line 11 `dt` translates `$filename` based on `%handler` functions.

The result will be:

```
 <b>The XML Down Translator</b>
   <i>J.J. Almeida</i>
   <i>J.C. Ramalho</i>
<br>
```

2.2 mkskel.pl: a program to generate XML::DT processors

In order to simplify the task of making XML::DT processors we have developed "mkskel.pl" - a program that generates a skeleton of a XML::DT processor for a target XML document. It has the peculiarity of being programmed with XML::DT, illustrating this way the use of XML::DT in another kind of applications.

The default action (actually the only one defined) makes a side-effect: it computes the list of elements used in the target xml file.

In the end `mkskel.pl` program writes a XML::DT processor associating a simple action to each element name found.

```
#!/usr/bin/perl
use XML::DT ;
my $filename = shift;
%xml=( '-default' => sub{$element{$q}=1; ""});

dt($filename,%xml);

print <<'END';
#!/usr/bin/perl
use XML::DT ;
my $filename = shift;

%handler=(
#    '-outputenc' => 'ISO-8859-1',
#    '-default'   => sub{"<$q>$c</$q>"},
END

for $name (keys %element){
     print "     '$name' => sub{\"\$q:\$c\"},\n";
}
```

```
17 print <<'END';
18 );
19 print dt($filename,%handler);
20 END
```

The default function in line 4 is called for each element in the XML file. This default action, just stores the element name `$q` in the `element` associative array. In fact a very simple perl statement, we are just creating a list of element names.

Whenever necessary, much more complex actions can be included in the processing functions.

In lines 6 to 20 a perl XML::DT program is written to the output. In lines 14 to 16 a simple processing line is written for each name in the `element` list (associative array). This simple processing line, can be changed by the user to meet more specific needs, generating a more specialized processor skeleton.

The output of `mkskel.pl art.xml` is:

```
#!/usr/bin/perl
use XML::DT ;
my $filename = shift;

%handler=(
#    '-outputenc' => 'ISO-8859-1',
#    '-default'   => sub{"<$q>$c</$q>"},
     'title' => sub{"$q:$c"},
     'author' => sub{"$q:$c"},
     'article' => sub{"$q:$c"},
     'abstract' => sub{"$q:$c"},
     'keyword' => sub{"$q:$c"},
);
print dt($filename,%handler);
```

2.3 Making proceedings end-page

Suppose that we have a set of papers and we want to generate the proceedings book with those papers. The proceedings could be defined as:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<proceedings>
<title>The XML Europe 99</title>
<chair>Pam</chair>
<abstract>
   Once upon a time in Granada ...
</abstract>
```

```
<article file="art2.xml"/>
<article file="art3.xml"/>
<article file="art1.xml"/>
</proceedings>
```

Now we can generate the proceedings by writing a proceedings' processor.

In order to make the example shorter we are going to discuss just the case of making the proceedings end page with the titles and the list of included papers.

Note that the papers are not copied in this document; the article empty element just contains an attribute named "file" with the name of the XML paper document.

The proceedings processor calls a paper processor to do the job.

```
 1  #!/usr/bin/perl
 2  use XML::DT ;
 3  my $filename = shift;

 4  %p_proc=(
 5       '-default'    => sub{"$c"},
 6       'proceedings' => sub{"Proceedings $c"},
 7       'abstract'    => sub{""},
 8       'article'     => sub{ dt($v{file}, %p_art) },
 9       'chair'       => sub{"Chair: $c"},
10  );

11  %p_art=(
12      '-default'     => sub{""},
13       'title'       => sub{"  $c"},
14       'author'      => sub{"    <i>$c</i>"},
15       'article'     => sub{"$c"},
16  );

17  print dt($filename,%p_proc);
```

The default action (line 5) just returns element content. The element abstract is ignored (line 7), and some syntactic sugar is added (lines 6 and 9).

In line 8, `$v{file}` is used to obtain the value of the attribute "file". When an element "article" is found `dt` function is called with the filename (from the attribute "file") and the paper processor `%p_art` similar to the example previously presented.

In this example we are showing how several processors can be coexist to process the same XML document enabling subdocument processing.

The generated output was:

```
Proceedings
```

```
The XML Europe 99
Chair: Pam
   The XML Parser
     <i>Clark Cooper</i>
     <i>Larry Wall</i>
   The expat tool
     <i>James Clark</i>
   The XML Down Translator
     <i>J.J. Almeida</i>
     <i>J.C. Ramalho</i>
```

2.4 Making a keyword index

In previous example, each paper had *keyword* tags. In this example we will compute a richer proceedings end-page by adding a keyword index:

```
 4  %p_proc=(
 5       ...
 6       'proceedings' => sub{ "Proceedings $c". mkKeyInd() },
 7       ...

11  %p_art=(
12       ...
13       'title'       => sub{ $tit= $c; "  $c"},
14       ...
16       'keyword'     => sub{ $ind{$c} .= "\n      $tit"; "";}
17  );

19  sub mkKeyInd { my $r="Index by keywords\n";
20    for $term (sort keys %ind){ $r .= "\n   $term $ind{$term}";}
21    $r
22  }
```

In line 13 a side-effect was added to save the title in `$tit` variable.

In line 16 we are building a keyword index as an association of keyword to a string containing the titles separated with new lines.

In line 6, we concatenate the previous solution with the result of a function `mkKeyInd()` defined in lines 19 to 22. `mkKeyInd()` returns a string containing the index text.

In this example we can see that is easy to mix simple side-effects in the processors in order to build other views of the document. This approach is similar to the attributed grammars view.

The generated output was:

```
Proceedings
```

```
The XML Europe 99
chair: Pam
   The XML Parser
      <i>Clark Cooper</i>
      <i>Larry Wall</i>
   The expat tool
      <i>James Clark</i>
   The XML Down Translator
      <i>J.J. Almeida</i>
      <i>J.C. Ramalho</i>
Index by keywords
   XML
      The XML Parser
      The expat tool
      The XML Down Translator
   expat
      The expat tool
   language processing
      The XML Down Translator
   perl
      The XML Parser
      The XML Down Translator
```

2.5 Context

Processing XML elements is often context dependent: the actions that should be performed are parent dependent. In order to be possible to write context dependent processing, two functions are provided:

```
ctxt(number)
inctxt(pattern)
```

`ctxt(1)` returns the name of the father element; `ctxt(2)` returns the name of the grand-father element.

`inctxt(pattern)` returns true if the pattern matches the context path string.

Suppose that the papers have sections with titles and contents. In order to have the correct end-page generation, some changes are necessary. Just the titles with parent "article" should be saved.

```
...
title => sub { if(inctxt('article'))
                  {$tit=$c; "  $c";}
               else
                  {""}
```

```
                }
    ...
```

or

```
    title => sub { if(ctxt(1) eq 'article')
    ...
```

3 The main algorithm

The algorithm that is presented in this section is a simplification of real one in order to be easier to read. A Haskell (functional) like notion is used.

`dt` function processes the tree resulting from parsing the file received as an argument.

```
dt(filename,processor)=
  let tree=Parse(filename)
  in process(tree,processor)
```

```
process(PCDATA(p), processor)        = p
process(element(e,sons), processor) =
   let args = concatenate( [ process(x,processor) | x <- sons] )
   in if(e in dom(process)) then  processor[e](args)
                            else  processor["-default"](args)
```

Processing a PCDATA text returns the text.
Processing an element is done by:

- processing its sons
- concatenating their results
- applying the corresponding processor function to the previous result

4 Conclusions and future work

XML::DT was design to do simple tasks.

Our experience with using and teaching XML::DT was good: it follows the natural structure of the documents. It is possible to write a XML processor with a very small perl program.

The ability of putting a XML processor in a single perl variable is powerful and enables natural sub-document processing through the coexistence of several processors in the same specification.

In this version DT returns a string. Some examples needed a different type of result which can be done with side-effects. Work is under progress to include primitives to compute non atomic (string) results based on DTD information.

References

[1] Omnimark Technologies, http://www.omnimark.com

[2] AIS Software, http://www.balise.com

[3] Thompson, H.;

[4] Megginson, D.; SGMLS.pm: a perl5 class library for handling output from the SGMLS and NSGMLS parsers; http://home.sprynet.com/ dmeggins/software.html.

[5] Cooper, C.; XML - Parser, a Perl interface to expat; http://www.netheaven.com/ coopercc/xmlparser/intro.html

[6] Cooper, C; Using The Perl XML::Parser Module; http://www.xml.com/xml/pub/98/09/xml-perl.html

[7] Clark, J.; expat - XML Parser Toolkit; http://jclark.com/xml/expat.html

[8] Clark, J.; SGML Parser; http://www.jclark.com

[9] http://www.perl.com

[10] http://www.unicode.org