HopTeX - Compiling HTML to LaTeX with CSS

Manuel Serrano

Inria Sophia Méditerranée
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex – France
Manuel.Serrano@inria.fr

Abstract

This article[1] presents HopTeX, a new application for authoring HTML and LaTeX documents. The content of the document is either be expressed in HTML or in a blending of HTML and a dedicated wiki syntax, for the sake of conciseness and readability. The rendering of the document is expressed by a set of CSS rules. The main originality of HOPTEX is to consider LaTeX as a new media type for HTML and to express the compilation from HTML to LaTeX by the means of dedicated style sheet rules.

HOPTEX can then be used to generate high quality documents for both paper printed version and electronic version. The online version of this paper is available at the HOPTEX web page. It can be read using a regular Web browser or using Smartphone browsers.

HOPTEX is implemented in HOP, a multi-tier programming language for the Web 2.0. This implementation extensively relies on two facilities generally only available on the client-side that HOP also supports on the server-side of the application: DOM manipulations and CSS server-side resolutions.

`http://hop.inria.fr/hop/weblets/homepage?weblet=hoptex`

1. Introduction

Many scientific publications, in particular in academia, are authored with TEX or LaTeX [10, 11]. This is a *batch* system where documents are actually disguised programs that, when executed, produce various output document formats including DVI or PDF.

Although the TEX programming language is Turing-complete, it is mostly exclusively used as a purely authoring declarative language. Being more than forty years old it lacks most modern features of programming languages: its syntax is difficult to parse, it supports no object-oriented features, and it offers a limited set of functions for interacting with the operating system. In consequence, programming in TEX requires a strong expertise that is repellent to many, although a small community of aficionados is able to use it beyond expectations (see for instance [6]). On the other hand TEX is still widely used because its rendering engine, coupled with the MetaFont tool, delivers high quality documents that hardly no contemporary typesetting system matches.

The most striking shortcoming of TEX/LaTeX is its inability to produce HTML. Since publishing on the web is nowadays mandatory, translators from LaTeX to HTML such as Latex2html or Hevea [13] have emerged. These tools have limitations because they offer few facilities for controlling the graphical rendering of the generated documents. This limitation comes from their inability to use CSS with the generated HTML documents because these lack HTML classes or HTML identifiers.

Other tools such as Skribe [5] and Scribble [4] follow the symmetrical path which consists in considering LaTeX as a target and no longer as a source. They attempt to improve LaTeX by providing a sane programming language used to generate the texts. They offer an *ad hoc* syntax that combines algorithmic constructs and text oriented markups. A program can generate LaTeX as well as HTML. These two systems are agnostic with respect to the generated format. As a consequence of this design choice, they adopt abstractions reflecting a least-common denominator of their target formats. That design choice also makes them difficult to use when fine grain tuning of the generated document is needed. This characteristic is shared by systems such as Texinfo or DocBook [16] that represent texts using a neutral syntax that can be either compiled to HTML, LaTeX, and even other formats.

Accommodating HTML as a regular data type in programming language is not new. DSSSL [8], the pioneer and LAML are two examples based on the Scheme programming language [9]. Other languages such as XDuce [7] or XQuery [2] extends this to XML. These languages are well suited for manipulating XML documents but they have no particular skill for authoring documents.

HOPTEX is a new system for authoring articles, reports, documentation, and books that follows yet another approach. It accepts as input either HTML or a compact wiki syntax that can be seconded by the expressions of the HOP programming language. It either produces web pages or LaTeX files. HOPTEX aims at combining the best of the two worlds: it generates HTML for using the modern interactive features of the web browsers and it generates LaTeX for producing high quality paper output. This approach enables HOPTEX to generate online documents that embed arbitrary HTML fragments such as videos, canvas, pictures, or interactive Ajax elements. It also enables HOPTEX to generate paper documents that rely on pre-existing LaTeX styles. HOPTEX generates regular LaTeX files so it is up to the user to include the correct proper statement in his document source. For instance, to accommodate the ACM style required by the conference, the head of the present paper contains the following:

[1] Work partially supported by the French ANR agency, grant `ANR-09--EMER-009-01`.

```
<tex:verbatim>
\documentclass[nocopyrightspace]{sigplanconf}
\usepackage{amsmath}
\usepackage{graphicx}
\usepackage{color}

\setlength{\pdfpagewidth}{8.5in}
\setlength{\pdfpageheight}{11in}
...
\maketitle
</tex:verbatim>
```

HOPTEX is implemented in HOP [14], a multi-tier programming language for the web. HOP offers features that dramatically simplify the implementation of HOPTEX. In particular, it constructs a server-side DOM for the HTML documents and it supports a server-side CSS resolver. These two features are extensively used to compile to LaTeX.

This presentation of HOPTEX is organized as follows. First, to let users unfamiliar with the HOP programming language understand this paper without consulting previous articles, the language is briefly presented in Section 2. Section 3 presents the main functionalities of HOPTEX. Section 4 shows how LaTeX is generated out of the initial HTML document. Section 5 shows the benefit HOPTEX users can expect from resorting to a full-fledged web programming language.

2. Background, the Hop programming language

HOP is a multi-tier programming language for the web which shares many characteristics with JavaScript. It belongs to the functional languages family. It relies on a garbage collector for automatically reclaiming unused allocated memory. It supports type annotations that let the compiler partially check types at compile-time. Types that cannot be inferred are check dynamically at runtime. It is fully polymorphic (*i.e.*, the universal identity function can be implemented). HOP has also several differences with JavaScript, the most striking one being its parenthetical syntax closer to HTML than to C-like languages. HOP is a full-fledged programming language so it offers an extensive set of libraries. It advocates CLOS-like object oriented programming [1]. Its main characteristic is that it fosters a programming model where a web application is conceived as a whole. For that, it relies on a single formalism that embraces simultaneously server-side and client-side of the applications. Both sides communicate by means of function calls and signal notifications. Server-side parts are compiled to a mix of bytecode or native code and client-side parts are compiled to JavaScript [12]. In the source code, a syntactic mark instructs the compiler about the location where the expression is to be evaluated.

When an URL is intercepted by a HOP server for the first time, the server *automatically* loads the associated program and the libraries it depends on. Programs first authenticate the user they are to be executed on behalf of and they check his permissions. In order to *load* or *install* the program on the client side, the server elaborates an abstract syntax tree (AST) and compiles it on the fly to generate a HTML/JavaScript document that is sent to the client. Here is an example of a simple HOP program that is started by browsing the URL `http://localhost/hop/hello`.

```
(define-service (hello)
   (<HTML> (<DIV> :onclick ~(alert "world!") "Hello")))
```

Contrary to HTML, HOP's markups (*i.e.*, ,`<HTML>` and `<DIV>`) are node *constructors*. That is, the service `hello` *elaborates* an AST whose compilation into HTML is delayed until the result of the request is transmitted to the client. This two phased evaluation process is strongly different from embedded scripting language such as PHP. The AST representing the GUI exists on the client as well as on the server. This brings flexibility because it gives the server opportunities to deploy optimized strategies for building and manipulating the ASTs as it lets DOM computations take place on the server-side of the application. This characteristic is extensively used for implementing HOPTEX.

3. HopTeX

This article not being a HOPTEX user manual only its prominent features are presented. HOPTEX documents are expressed in HTML. However, because HTML concrete syntax is verbose it is cumbersome to manipulate for the user. HOPTEX therefore proposes an alternative wiki syntax that can be used in conjunction of HTML. It is expected that this syntax will be preferred by users so it is first presented in this section. Secondly, it is shown how the wiki syntax and the full-fledged HTML syntax can be blended inside documents.

3.1 The surface syntax

HOPTEX syntax is stratified: the *surface* syntax is used to typeset input texts, the *deep* syntax, which coincides with the syntax of the HOP expressions, is used to embed complex HTML trees in the document. The surface syntax is inspired by most popular wiki syntaxes and in particular by MediaWiki[2] and CreoleWiki[3]. It allows authors to express a subset of HTML in a *concise* and *visual* way. For instance, tags for strong and emphasize are `**` and `//` which are considered by some more intuitive and more compact than the corresponding HTML tags. For instance, the following HOPTEX input text:

```
HOP wiki supports **strong**, //emphasize//,
__underline__, and ++mono space++. These
can be **__combined__**  **//anyhow//**.
```

is rendered as:

HOP wiki supports **strong**, *emphasize*, underline, and `mono space`. These can be **combined** ***anyhow***.

The surface syntax supports sections (`==`), paragraphs (`~~`), verbatim texts (lines beginning with two white spaces), tables (lines beginning by either `^` or `|`), lists (lines beginning with two whitespaces followed by either a `*` or `-` character), or other classical block constructs that are separated one entry from another by two blank lines. For instance, the following table:

```
| This    |  is  ^    a table ^
```

produces the following result:

This	is	**a table**

The delimiter `^` introduces table head while the delimiter `|` introduces regular table cells. This explains why the words "a table" is rendered with a bold font in the example above.

HOPTEX supports mathematical expressions which are introduced by the `$$` delimiter. Inside this delimiter HOPTEX borrows the syntax of TeX whose syntax for mathematics is deemed expressive and compact. Mathematical expressions are compiled to MathML on the fly. For instance:

```
  * $$\prod_n^m \lim_{n \rightarrow \infty}
 x = 0$$
  * $$\overbrace{\overline{x}^{2} + 1}$$
  * $$(n+1)^2\quad \sqrt{1-x^2}\quad
 \overline{w+\bar z} \quad p^{e_1}_1$$
```

[2] `http://www.mediawiki.org`

[3] `http://www.wikicreole.org`

produces:

- $\prod_n^m \lim_{n\to\infty} x = 0$
- $\overbrace{\bar{x}^2 + 1}$
- $(n+1)^2 \quad \sqrt{1-x^2} \quad \overline{w+\bar{z}} \quad p_1^{e_1}$

Links and anchors are syntactically similar to those of MediaWiki but extended to support citations, references, and footnotes that are introduced by using a dedicated protocol (`bib:` for citations, `section:` for sections, ...). For instance:

```
Links refer to URLs such as
++[[http://www.inria.fr]]++.
They may also refer to sections or
bibliographic entries  such as:
HopTex is described in Section
[[section://HopTeX]].
```

produces:

Links refer to URLs such as `http://www.inria.fr`. They may also refer to sections or bibliographic entries such as: HopTex is described in Section 3.

3.2 The deep syntax

The surface syntax trades *completeness* for *compactness*. That is not all HTML trees can be represented using the surface syntax. For such trees, the *deep* syntax is used. The escaping sequence of the deep syntax is `,(`. When the HOPTEX parser reads such a prefix, it reads the rest of the expression using the regular HOP parser, evaluates the expression, and inserts the result in the tree. For instance:

```
The //deep// escape sequence is
,(<TT> ",("). It can be used to insert
HTML trees such as ,(<KBD> "C-x s"). The
++<WIKI>++ markup is used to
,(<SPAN> :style "color: darkblue"
   (<WIKI> [enter the //surface// syntax
from the //deep// syntax])).
```

produces:

The *deep* escape sequence is `,(`. It can be used to insert HTML trees such as `C-x s`. The `<WIKI>` markup is used to enter the *surface* syntax from the *deep* syntax.

4. Generating TeX

Wiki syntaxes such as the HOPTEX surface syntax are designed to express a subset of HTML concisely. As such, they are easy to translate into HTML. They are far less obviously translated into TEX. This translation is described in this section.

Observation 1: TEX/LATEX (henceforth LATEX) and HTML are not isomorphic. HTML is more flexible and more compositional. For instance a HTML `TABLE` might contain `PRE` elements while LATEX refuses `verbatim` environments inside a `tabular`. Consequently not all HTML documents, and thus HOPTEX documents, can be automatically compiled into LATEX.

Facing this problem, two obvious solutions emerge: either reduce the expressiveness to HOPTEX to the least common denominator of HTML and LATEX, or treat HTML parts that have no LATEX equivalent specially. We have considered the intersection of the two languages too small so we have adopted the latter solution. In consequence, from time to time, HOPTEX users have to specify explicitly how to compile some part of the text into LATEX. However, we have worked hard to minimize the number of occurrences of such situations and we have worked even harder to provide convenient means for expressing these *ad-hoc* compilation schemas.

Observation 2: Cascading Style Sheets (henceforth CSS) [3] effectively separate the structure of a document from its rendering. If compiling HTML into LATEX is possible roughly equivalent to *rendering* HTML into LATEX, then, CSS could probably be used for that compilation.

Consider our previous example using bold-face fonts and italic and consider what happens if we ask a web browser to render them using the following CSS rules:

```
strong:before { content: "{\\textbf{"; }
em:before { content: "{\\emph{"; }
strong:after, em:after { content: "}}"; }
```

The browser will display the following document

```
HOP wiki supports {\textbf{bold}},
{\emph{italic}}...
```

which is *almost*[4] a LATEX compilation.

The HOPTEX compilation relies on CSS in a principled manner where the compilation rules are expressed as CSS rules. In addition to simplicity, using CSS also brings flexibility because it let users provide their own compilation rules in their own CSS files that can override the default compilation strategy.

4.1 CSS driven compilation

The browser cannot be used to implement the compilation as a simple HTML rendering for two reasons. First, the browser cannot save the rendered text. Second, some compilation rules are more complex than merely adding a prefix and a suffix. For instance, in HTML, `pre` elements are regular blocks that only differ from paragraph by not collapsing white spaces and by breaking lines at `newline` character positions and by using a dedicated font. LATEX has nothing similar. The `verbatim` environment has the same behavior for justification and line breaks but considers markups as plain texts. Extensions such as `alltt` approach `pre` but all have incompatibilities. In consequence, HTML `pre` elements have to be treated specially when compiled to LATEX.

HOPTEX relies on *server-side* CSS processing. It resorts to the HSS [15] compiler which is included in the HOP development environment [14]. Amongst other features, HSS contains a *parser* that builds abstract syntax trees and a *resolver* that matches rules against HTML elements.

When a HOPTEX input text is to be compiled into LATEX, the *surface* syntax is first parsed to produce a full-fledged server-side DOM representation of the HTML document. The elements of this tree are matched against CSS rules which govern the compilation into LATEX. The extra `tex` keyword can be used in CSS `@media` rules to specify rules that are only applicable to the LATEX compilation.

The rest of this section presents the details of the compilation. The algorithm is expressed by 4 HOP functions. We deem the HOP language sufficiently high level to be used as an abstract notation for describing these algorithms. Readers unfamiliar with functional programming will probably find some details of the implementation obscure. We hope they will still be able to grasp the general intuition of the algorithms.

The service `hoptex/tex` implements the entry point of the compiler. It accepts two parameters, the URL of the source file to be compiled and the name of the target file. The service first builds a server side DOM for the document (using the library function `wiki-file->dom`). Then it loads the CSS style sheets imported in the DOM tree and invokes the `xml->tex` function.

[4] Almost compilation only because apart from using *cut-and-paste* there is no means to save the result of this compilation.

```
(define-service (hoptex/tex url dest)
  (let* ((doc (wiki-file->dom url))
         (hd (dom-get-elements-by-tag-name doc "head"))
         (css (map tex-load-hss (links-of-head hd))))
     (call-with-output-file dest
        (lambda (op) (xml->tex doc css op)))))
```

The function `xml->tex` is in charge of compiling one node of the DOM tree into one LaTeX element. The parameter `node` is the node to be compiled, the parameter `css` is the opaque data structure representing the CSS rules, and the last parameter `p` is the output port where to write the result of the compilation. Numbers are written in the target file without modification; strings are escaped, that is, all special LaTeX characters are protected against interpretation (the function `tex-string` is in charge of this task); lists are recursively processed; and XML nodes are treated specially by the function `xml-elements->tex` which is given in Figure 1.

```
(define (xml->tex node::obj css::obj p::output-port)
   (cond
      ((string? node)
       (display (tex-string node) p))
      ((number? node)
       (display node p))
      ((list? node)
       (for-each (lambda (o) (xml->tex o css p)) node))
      ((xml-element? node)
       (xml-element->tex node css p))))
```

Compiling a XML element is decomposed in 7 steps.

1. *Compute node style*. It is computed by the library function `css-get-computed-style`. If no style is found then the compilation simply compiles recursively the children of the node.
2. *Check if the element is visible*. The style may make an element invisible if it contains declarations such as `display: none`. Invisible elements are ignored by the compiler.
3. *Compile the prelude*. The prelude is computed using the tag of the node and the elements of the style.
4. *Compile the before attribute*. The "before" attribute is string of characters that has to be inserted before the current element. It is handled by the function `xml-style->tex`. For instance, the default "before" attribute of the HTML `em` nodes is the string "`{\em{`". The "before" attribute can be customized by users while the prelude is hardwired in HopTeX.
5. *Compile the body of the node*. This involves two cases. If the CSS style contains a dedicated compiler for the node, use that compiler. Otherwise, recursively compiles the children nodes.
6. *Compile the after attribute*. The "after" attribute is symmetrical to the "before" attribute. It closes the LaTeX environment opened in the "before" attribute.
7. *Compile the postlude*. The postlude is symmetrical to the prelude. It mostly consists in closing the environment opened in the prelude. For instance, if the prelude as emitted "`{\small{`", the postlude emits "`}}`".

The function `xml-style->tex`, not given here, is a trimmed down version of `xml-element->tex` that is in charge of processing the content strings of the `or` attributes.

4.2 Examples

In this section we present a few examples of compilation and we show how users can change the generated LaTeX rendering by providing additional CSS rules.

4.2.1 Example 1, a simple compilation

Assuming the CSS rules given in Section 4, let us study the compilation of the following text:

```
A **strong //and emphasized//** text
```

First, the server parses the text and translates it into HTML. Along this process, it builds a DOM representation of the following tree:

```
<DIV> A <STRONG>strong <EM>and emphasized
</EM></STRONG> text</DIV>
```

The compiler has to compile the `DIV` elements which has three children: the string "`A`", the `DIV` containing the "`STRONG...`" elements, and the string "`text`". Since the `DIV` element has no style attached to it then its compilation consists in a simple traversal of the tree. The first string is written as is. Then comes the compilation of the `STRONG` and `EM` elements. These ones have styles that specify a "before" and "after" strings that are inserted in the generated LaTeX output. The result of the compilation is:

```
A {\textbf{strong{\emph{and emphasized}}}} text
```

4.2.2 Example 2, adding user rules

A user wanting to emphasize even more texts which are under a `STRONG` and a `EM` elements could use his own CSS rule such as (remember that the `>` CSS operator filters direct descendant of a node):

```
strong > emph {
  text-decoration: underline;
}
```

This changes the compilation of the `EM` nodes whose parents are `STRONG` nodes. It adds the rule `text-decoration: underline` to the style computed by the `css-get-computed-style` that enriches the default compilation of `EM` elements. The generated LaTeX code becomes:

```
A {\textbf{strong{\emph{\underline{and
emphasized}}}}} text
```

4.2.3 Example 3, designating elements

As with HTML, CSS rules for HopTeX can be used to change the compilation of individual nodes. A simple way to achieve this is to assign identifiers to nodes and use these identifiers in the rules. Wiki tags used by HopTeX accept identifier and class declarations. They are given by suffixing the tag with `:id@class`. For instance, one may write:

```
~~:p1@note This is a note.
```

which defines a paragraph named `p1` that belongs to the class `note`. Identifiers and classes can be used in rules such as:

```
p.note:before { content: "Note:"; font-style: italic; }
@media tex { #p1 { font-size: 70%; } }
```

The "`p.note:before`" rule applies to all rendering engines. So in particular to the LaTeX code generator that adds the italicized version content before the paragraph. The "`#p1`" rule only applies to the LaTeX compilation because protected by a "`@media tex`" rule. It instructs the code generator to use tiny font the paragraph "`#p1`" that will be compiled as:

```
\begin{tiny}{\textit{Note:} This is a note.\end{tiny}}
```

4.3 Three particular cases

As mentioned in Section 4, resorting to "before" and "after" attributes of CSS style suffice to compile most HTML elements. However, for a few of them, inserting a prefix and a suffix is not enough.

```
(define (xml-element->tex node::xml-element css p)
  ;; step 1: compute the style
  (let ((style (css-get-computed-style css node)))
    (if (css-style? style)
        ;; step 2: check visibility
        (when (css-visible? style)
           (xml-element-visible>tex node css p style))
        ;; step 1b: plain recusive compilation
        (xml->tex (xml-element-body node) css p))))

(define (xml-element-visible->tex n css p style)
  (with-access::css-style style (after before)
    (let ((texc (style->tex (xml-element-tag n) style))
          (css-proc (css-style-get-attribute style 'proc)))
      ;; step 3: tex prelude
      (for-each (lambda (t) (display (car t) p)) texc)
      ;; step 4: style :before
      (when (css-style? before)
         (xml-style->tex before css p))
      ;; step 5: body compilation
      (if (procedure? css-proc)
          ;; step 5b: a dedicated compiler is used
          (css-proc n css p)
          ;; step 5c: a simple recursive descent is used
          (xml->tex (xml-element-body n) css p))
      ;; step 6: style :after
      (when (css-style? after)
         (xml-style->tex after css p))
      ;; step 7: tex postlude
      (for-each (lambda (t) (display (cdr t) p)) texc))))
```

Figure 1. Compiling XML elements.

The current HOPTEX version makes a special case for exactly 4 elements, namely `IMG`, `PRE`, `TABLE`, and `A`. We present the compilation of the first three in this section. The compilation of `A` is delayed to Section 5.1.

When CSS prefixes are not enough, an *ad hoc* compilation function can be defined. These functions are declared in the rules as the value of the HOPTEX specific `proc` property. They are HOP functions that HOPTEX calls with three parameters: the `node` to be compiled, the current `css` rule set, and the output `port` where the result should be written. Let us illustrate these compilation function on three examples.

4.3.1 Compiling images

Images are inserted in the text with either the regular `IMG` markup or with the wiki syntax `{{...}}` as in:

```
{{screenshot.png|a screenshot}}
```

Images are compiled in LATEX into a `includegraphics` environment in which image resizing is expressed as a ratio of the line width. The HOPTEX function `xml->tex-img` is in charge of this translation. It computes the LATEX size of the image. If no width is specified for a image, the generated LATEX image spans over the whole line. If a width is given, the percentage string is converted into a floating point value in the range $[0..1]$, which is concatenated to the string `\linewidth`.

```
(define (xml->tex-img node::xml-img css p)
  (fprintf p "\\includegraphics[width=~a]{~a}"
     (let ((w (node-computed-style node :width css)))
       (if (string? w)
           (let ((m (pregexp-match "^([0-9]+)%$" w)))
              (if (not m)
                  ;; a string such as "10em"
                  w
                  ;; a percentage
                  (format "0.~a\\linewidth" (cadr m))))
           "\\linewidth"))
     (dom-get-attribute node "src")))
```

The default HOPTEX rule for compiling images is:

```
img { width: 80%; proc: $xml->tex-img; }
```

The dollar sign before the `xml->tex-img` is a syntactic annotation that tells the CSS parser that the following expression is not a literal but a value of the HOP language. The compilation of the image given above using the previous CSS is:

```
\includegraphics[width=0.8\linewidth]
  {screenshot.png}
```

4.3.2 Compiling pre-formatted blocks

As noted in Section 4, HTML `PRE` elements have no direct LATEX counterpart. To compile them, HOPTEX generates a full line wide `tabular` nested in a `texttt` environment, and it replaces all white spaces with the explicit command `\ ` that forces LATEX to introduce plain blank characters. The implementation of this function is as follows:

```
(define (xml->tex-pre node::xml-pre css p)
  (with-access::xml-pre node (body)
    (display "\\noindent\\texttt{" p)
    (display "\\begin{tabular*}{\\linewidth}" p)
    (display "{l@{\\extracolsep{\\fill}}}\n" p)
    (let loop ((b body))
       (cond
          ((string? b)
           (let ((s (tex-string b)))
              (display
               (string-substitute s " \n" "\\ " "\\\\\n")
               p)))
          ((pair? b)
           (for-each loop b))
          (else
           (xml->tex b css p))))
    (display "\\end{tabular*}}\n" p)))
```

The CSS rule that accommodates this compilation scheme is:

```
pre {
  font-size: small;
  proc: $xml->tex-pre;
}
```

4.3.3 Compiling tables

HTML tables and LaTeX tabulars have nearly orthogonal designs. HTML tables tunings are expressed on a per-cell basis while LaTeX tables are configured on a per-column/per-row basis. In consequence, compiling HTML tables into LaTeX tabulars is inherently *ad hoc*. The default HOPTEX compilation flushes left cells and includes no rule at all. The function `xml->tex-table` first counts the number of columns in order to generate the LaTeX columns declaration. Then, each row of the table is compiled with the function `xml->tex-tr` that separates each element with the `&` sign and that inserts an end of line delimiter after each row.

```
(define (xml->tex-table el::xml-table css p)
   (define (count-columns obj)
      (define (tr-count-columns obj)
         (length (xml-element-body obj)))
      (apply max
        (map tr-count-columns (xml-element-body obj))))

   (fprintf p "\\begin{tabular}{~a}\n"
            (make-string (count-columns el) #\l))
   (xml->tex (xml-element-body el) css p)
   (display "\\end{tabular}\n" p))

(define (xml->tex-tr el::xml-tr css p)
   (with-access::xml-element el (body)
      (if (null? body)
          (display "\\\\\n" p)
          (let loop ((body body))
             (xml->tex (car body) css p)
             (if (null? (cdr body))
                 (display " \\\\\n" p)
                 (begin
                    (display " & " p)
                    (loop (cdr body))))))))
```

The default CSS rules for table are as follows:

```
table { proc: $xml->tex-table; }
tr { proc: $xml->tex-tr; }
th:before { content: "{\\textbf{\\textsf{"; }
th:after { content: "}}}"; }
```

In addition to connecting the two functions above to the `TABLE` and `TR` elements, it also configures `TH` elements to mimic their HTML default appearance. Provided with these declarations, the table example given Section 3.1 is compiled as:

```
\begin{tabular}{lll}
This & is & {\textbf{\textsf{a table}}} \\
\end{tabular}
```

5. A full-fledged programming language

In this section we illustrate the benefits of using a full-fledged programming language in HOPTEX by presenting two extensions. We show how to manage bibliographic references and how to delegate the placement of floating elements to CSS rules.

5.1 Accommodating bibliography

Bibliography citations are treated by HOPTEX as a special kind of external hyperlinks. Consistently, the wiki syntax is augmented with the new `bib://` protocol that accommodates citations which then look like:

```
[[bib://knuth:tex86 lamport:latex86]]
```

Because the BibTex format is widely used, it has been found appropriate to make it directly usable in HOPTEX. For that, a full BibTex parser has been implemented in HOPTEX. When a document is to be processed, the BibTex bibliography database is then parsed and stored in a hash table. Then the DOM is traversed and all citations are adjusted. For the sake of the example, here is the code in charge of this traversal:

```
(define (citation? e)
  (when (xml-element? e)
    (with-access::xml-element e (attributes)
      (let ((href (xml-get-attribute :href attributes)))
         (and href
           (string? (xml-attribute-value href))
           (string-prefix? "bib://"
                           (xml-attribute-value href)))))))

(define (dom-get-citations expr)
  (filter citation?
          (dom-get-elements-by-tag-name expr "a")))
```

It uses regular DOM functions, that in HOP are also available on the server-side of the applications, to retrieve all the link elements (`A` HTML elements) whose links are prefixed by the "`bib://`" string.

5.2 Placing floats

Placing floating elements with LaTeX, is a nightmare that we have all lived once. Directives such as `htbn` are supposed to instruct the layout algorithm but they constantly fail. More strict directives have been added such as `!H` but in practice they show similar results. The only effective solution to trick the internal TeX algorithms consists in moving the floating elements in the source text back and forth. In addition to be painful and error prone this idiosyncratic behavior has an important drawback when a single source is used to generate LaTeX and HTML document. Since the web browser does not move float elements, the figures moved for LaTeX appear as randomly placed in the HTML version.

Because HOPTEX generates LaTeX documents from HTML specifications, we have an opportunity to improve over the previously described solution. Instead of moving the floating elements in the source text, HOPTEX moves them only in the generated LaTeX target accordingly to configurations expressed in CSS rules. For instance, one may write:

```
@media tex {
  #float1 {
    with: 100%;
    column-count: 2;
    float: -350;
  }
}
```

```
(define (move-float-backward! node offset)
   (let loop ((o offset)
              (prev node))
      (if (= o 0)
          (dom-insert-before!
           (dom-parent-node prev) node prev)
          (loop (- o 1) (dom-previous-node prev doc)))))

(define (dom-previous-node node doc)
   (let ((sibling (dom-previous-sibling node)))
      (if (not sibling)
          (dom-parent-node node)
          (dom-last-node sibling))))

(define (dom-last-node node)
   (let ((l (dom-child-nodes node)))
      (if (pair? l)
          (let ((n (car (last-pair l))))
             (if (xml-text-element? n)
                 n
                 (dom-last-node n)))
          node)))
```

Figure 2. Moving elements backward in a DOM tree.

which means that the float element named "`float1`" has to be moved 350 elements upward in the DOM tree.

Prior to generating LaTeX code the DOM tree is thus traversed to inspect all floating elements that have a `float` style attribute attached. Such elements are moved backward when the value is negative and forward when positive. The source code for moving a node in the tree is traditional DOM programming. It is given in Figure 2.

6. Conclusion

HopTeX is an operational system. It has already been used to write a couple of articles in addition to the present one. The whole implementation counts less than 4KLOC lines of Hop code and 1KLOC of Css rules. Such a compactness is possible only because it extensively uses the features offered by the Hop programming language: high level of abstractions supported by functional values, object-oriented support, full polymorphism, DOM server-side manipulation, Css server-side resolution, and builtin parsing facilities. HopTeX is free software released under the GPL license. It is available from the Hop web page.

References

[1] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common lisp object system specification. In *special issue*, number 23 in Sigplan Notices, Sept. 1988. URL `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/-ai-repository/ai/html/cltl/cltl2.html`.

[2] W. W. W. Consortium. XQuery 1.0: An XML Query Language. Technical Report REC-xquery-20070123/, W3C Recommendation, Jan. 2007.

[3] W. W. W. Consortium. Cascading Style Sheets level 2 revision 1 CSS2.1 Specification. Technical Report CR-CSS2-20090423, W3C Recommendation, Apr. 2009.

[4] M. Flatt, E. Barzilay, and R. B. Findler. Scribble: closing the book on ad hoc documentation tools. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 109–120, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596569. URL `http://www.cs.utah.edu/plt/publications/-icfp09-fbf.pdf`.

[5] E. Gallesio and M. Serrano. Skribe: a Functional Authoring Language. *Journal of Functional Programming*, 2005.

[6] A. Greene. BASIX – An Interpreter Written in TeX. *TUGBoat*, 11(3):381–392, 1990. URL `http://www.tug.org/-TUGboat/Articles/tb11-3/tb29greene.pdf`.

[7] H. Hosoya and B. Pierce. Xduce: a Typed XML Processing Language. In *In Proc. of Workshop on the Web and Data Bases (WebDB*, pages 226–244. Springer-Verlag, 2000.

[8] ISO/IEC. Information technology, Processing Languages, Document Style Semantics and Specification Languages (dsssl). Technical Report 10179:1996(E), ISO, 1996.

[9] R. Kelsey, W. Clinger, and J. Rees. The Revised(5) Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), Sept. 1998. URL `http://www.inria.fr/-mimosa/fp/Bigloo/doc/r5rs.html`.

[10] D. Knuth. *The TEXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Readings, Massachusetts, USA, 1986.

[11] L. Lamport. *LaTeX - a Document Preparation System*. Addison-Wesley, Readings, Massachusetts, USA, 1986. ISBN 0-201-15790-X.

[12] F. Loitsch and M. Serrano. *Trends in Functional Programming*, volume 8, chapter Hop Client-Side Compilation, pages 141–158. Seton Hall University, Intellect Bristol (ed. Morazán, M. T.), UK/Chicago, USA, 2008. ISBN 978-1-84150-196-3.

[13] L. Maranget. Hevea, un traducteur de LaTeX vers HTML en caml. In *Actes des 10e Journés francophones des langages applicatifs*. INRIA, 1999.

[14] M. Serrano. The HOP Development Kit. In *proceedings of the Seventh* ACM sigplan *Workshop on Scheme and Functional Programming*, Portland, Oregon, USA, Sept. 2006.

[15] M. Serrano. HSS: a Compiler for Cascading Style Sheets. In *10th* ACM Sigplan *Int'l Conference on Principles and Practice of Declarative Programming (PPDP)*, Hagenberg, Austria, July 2010.

[16] N. Walsh and L. Muellner. *DocBook: The Definitive Guide*. O'Reilly, Oct. 1999. ISBN 156592-580-7.