

Authoring dynamic websites with SXML

Peter Bex

February 8, 2007

1 Introduction

There are roughly two ways of dynamically generating websites. One way is the PHP way (or Perl, Ruby, etc). This means you simply write some HTML code and sprinkle code with side-effects in between. There are clear disadvantages to this. For example, operating on fragments of code must be done on the string-level, which is too low to do meaningful post-processing without writing ad-hoc HTML parsers. This also has the disadvantage that malicious or obnoxious HTML and scripts can be inserted relatively easy in the output by any potential attackers of your site, unless you take great care to escape HTML characters.

The other way is to use XML. Then you need to learn a number of different XML technologies like XSL, which includes XSLT and XPath or XQuery. On top of that, you still need to use a scripting language to express your business logic (XExpr, or any other scripting language like PHP). XML is also quite hard to read for a human being because of its verbosity. Any Scheme hacker who has done some moderate to heavy web programming will be annoyed by this state of affairs. Why can't we just use one tool to do it all? Well, we can!

By using SXML instead of these other technologies, you can use your existing knowledge of Scheme and a handful of procedures that can assist you in transforming XML in a completely functional way. Another advantage is that if you happen to have some existing XSL code, you do not have to discard it. You can simply take that code and feed it XML output from your SXML code without any problems.

There is quite a bit of information available at the [SSAX project page](#), but in my opinion it's quite fragmented and too academic. That's why I decided to write this hands-on tutorial. This tutorial is aimed at people who have never worked with SXML. It is assumed the reader is familiar with XHTML and has a working knowledge of Scheme. No knowledge of the corresponding XML technologies is assumed, but it may make it easier for you to understand. If you do not know Scheme yet, you may want to check out <http://www.schemers.org> to see what it's all about.

2 What is SXML?

SXML is simply a way to write XML as s-expressions. The official specification for SXML can be found at <http://okmij.org/ftp/Scheme/SXML.html>. A simple XHTML page looks like this:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
  <head>
    <title>An example page</title>
  </head>
  <body>
    <h1 id="greeting">Hi, there!</h1>
    <p>This is just an &gt;&gt;example&lt;&lt; to show XHTML &amp; SXML.</p>
  </body>
</html>
```

We can translate this to SXML by hand¹ and obtain the following:

```
(html (@ (xmlns "http://www.w3.org/1999/xhtml")
         (xml:lang "en") (lang "en")))
  (head
   (title "An example page"))
  (body
   (h1 (@ (id "greeting")) "Hi, there")
   (p "This is just an >>example<< to show XHTML & SXML.")))
```

Each element's tag pair is replaced by a set of parentheses. The tag's name is not repeated at the end, it is simply the first symbol in the list. The element's contents follow, which are either elements themselves or strings. There is *no* special syntax required for XML attributes. In SXML they are simply represented as just another node, which has the special name of `@`. This can't cause a name clash with an actual "`@`" tag, because `@` is not allowed as a tag name in XML. This is a common pattern in SXML: Anytime a tag is used to indicate a special status or something that is not possible in XML, a name is used that does not constitute a valid XML identifier.

We can also see that there's no need to "escape" otherwise meaningful characters like `&` and `>` as `&` and `>` entities. All string content is automatically escaped because it is considered to be pure content, and has no tags or entities in it. This also means it is much easier to insert autogenerated content and there is no danger that we might forget to escape user input when we display it to other users (which could lead to all kinds of nasty cross-site scripting attacks or other annoyances).

¹If we had translated the XHTML to SXML with a parser like SSAX, we'd end up with a slightly different structure, because it would interpret and encode the namespace information differently. To keep things simple, we'll just treat namespaces as simple attributes here.

3 SXML for websites

Now we know how to translate any X(HT)ML document to SXML, let's see how we can write SXML that gets translated to XHTML. The following illustrates the typical pattern we'll see a lot when generating websites:

```
(define document
  '(html (@ (xmlns "http://www.w3.org/1999/xhtml")
            (xml:lang "en") (lang "en")))
    (head
      (title "An example page"))
    (body
      (h1 (@ (id "greeting")) "Hi, there")
      (p "This is just an >>example<< to show XHTML & SXML."))))

(SRV:send-reply (pre-post-order document universal-conversion-rules))
```

The call to `SRV:send-reply` has the *side-effect* of displaying the HTML to the current output port so if you want it in a string you'll have to explicitly capture the current output port (eg, with `with-output-to-string` or some other implementation-specific procedure).

The procedure `pre-post-order` is the core of SXSLT. Right now we've only used it as a translator from generic SXML to something `SRV:send-reply` can output. If you just try to run `(SRV:send-reply document)`, you'll see the output is some kind of dumb concatenation of the flattened SXML tree. What `pre-post-order` does here is transform the SXML tree to some semi-flattened form of the SXML that can be concatenated so an XML string can be created by `SRV:send-reply`. The `universal-conversion-rules` are rules that tell it how it can do that. Don't worry if you don't understand this yet. We'll look at `pre-post-order` in much more detail in a few moments.

4 Semantic content

If you would only use the information above, you'd already have a very useful tool at your disposal. You can view any XML tree as a simple Scheme list. This means that any operation you can perform on lists, you can perform on SXML as well. A simple but useful example is when we would like to describe our pages in a more semantic way. For example, we would like to be able to write the following:

```
(define semantic-page
  '(page "Welcome to my homepage"
    (navigation)
    (greeting "Hi there")
    (p "This is a nice example page")
    (footer)))
```

We could use the same structure in every page. Actually, if every single page has a navigation and footer, we could even leave those out. We can see how this is a much more semantic way to describe our page. To actually transform this to valid XHTML, we could use the following code (which could be common code we include in all pages in our site):

```
(define (translator content)
  (cond
    ((null? content) '())
    ((list? (car content))
     (cons (translator (car content))
           (translator (cdr content)))) ;; Recurse down into lists
    ((eq? (car content) 'page)
     '(html (@ (xmlns "http://www.w3.org/1999/xhtml")
              (xml:lang "en") (lang "en"))
            (head (title ,(cadr content)))
                (body ,(translator (caddr content)))))
    ((eq? (car content) 'greeting)
     '(h1 (@ (id "greeting")) ,(cadr content)))
    ((eq? (car content) 'navigation)
     (cons
      '(ul
        (li (a (@ (href "home")) "homepage"))
          (li (a (@ (href "about")) "about this site"))
          (li (a (@ (href "contact")) "contact us")))
        (translator (cdr content))))
      ((eq? (car content) 'footer)
       '(p "Copyright (c) 2007"))
      (else (cons (car content) (translator (cdr content))))))

(define document (translator semantic-page))

(SRV:send-reply (pre-post-order document universal-conversion-rules))
```

I'm sure you'll agree this explicit rewriting of the SXML tree with custom code is not exactly fun. We'd like to have some kind of generalised way to do these rewrites, without having to explicitly write the behaviour every time. In other words, we'd like to define our transformations in a sort of *stylesheet* DSL. This is exactly what SXSLT is. We can write the above as follows:

```
(define my-rules
  '((page . ,(lambda (tag page-title . contents)
              '(html (@ (xmlns "http://www.w3.org/1999/xhtml")
                      (xml:lang "en") (lang "en"))
                    (head (title ,page-title))
                        (body ,contents))))
    (navigation . ,(lambda (tag)
```

```

        (ul
          (li (a (@ (href "home")) "homepage"))
            (li (a (@ (href "about")) "about this site"))
              (li (a (@ (href "contact")) "contact us")))))
(greeting . ,(lambda (tag str)
               '(h1 (@ (id "greeting")) ,str)))
(footer . ,(lambda (tag)
              '(p "Copyright (c) 2007")))
(*text* . ,(lambda (tag str) str))
(*default* . ,(lambda x x)))

(SRV:send-reply (pre-post-order
                 (pre-post-order semantic-page my-rules)
                 universal-conversion-rules))

```

Not only is the SXML shorter to write and less error-prone, but it is also clearer what is happening. Every high-level “tag” we defined is listed on the left, and the transformation code to run on that tag is shown on the right part. If you would like to take a look at the generated SXML code, do the following: `(pre-post-order semantic-page my-rules)`

4.1 Slowing down a bit

Let’s look at what happens here in more detail by investigating one rule up close:

```

(greeting .
 ,(lambda (tag str)
   '(h1 (@ (id "greeting")) ,str)))

```

The `pre-post-order` procedure walks the SXML tree almost in the same way our custom code did. The custom code simply looked at every element in the tree to see if it matched one of the expected symbols. But `pre-post-order` actually only looks at tags, ie the first symbol of a sublist. If the first rule does not match, it looks at the next rule, much like our custom code. If it finds a match for the tag, the tag name and all of its childnodes are passed to the transformation procedure as arguments. If there are no matches at all, the `*default*` rule is applied, which in this case leaves the content untouched. The `*text*` rule is applied to all leafnodes (ie, non-list nodes, which can be strings or symbols among other things). More about these special rules later.

In our case, the `greeting` element has only one element under it, the `greeting`’s text. This is put inside a `h1`. If we would like the name of the page to be printed smaller, we could simply modify this rule and every page would have its name printed smaller. It would also allow us to attach an id or class to it so we can target it with CSS for further styling.

If we look at the SXML code again for a second, we see that the `greeting` element looks very much like a procedure call to the `lambda` defined above:

```
'(greeting "Hi there")
```

The only difference is that the `lambda` accepts one more argument: the tag's name. This can be useful if you use the same procedure for several rules (or for a `*default*` rule).

5 Tree traversal methods

We have only seen part of `pre-post-order`'s power. The procedure is called that way because there are two different “orders” in which one can traverse an SXML-tree: Inside-out or outside-in. Let's look at another example:

```
(define counter
  '(child-count (children)))

(define counting-rules
  '((child-count .
    ,(lambda (tag children)
      '(kids ,(length children) ,children)))
    (children .
    ,(lambda (tag) ;; Just create 10 child tags
      (list-tabulate 10 (lambda _ '(child)))))
    (*text* .
    ,(lambda (tag str) str))
    (*default* .
    ,(lambda x x)))

(pre-post-order counter counting-rules)
```

This is a simple set of rules. The `children` rule generates 10 `child` elements. The `child-count` rule simply counts its children and puts the number in front of them. The question is now: Will it count 1 or 10? What it prints depends on whether `pre-post-order` traverses the tree pre-order or post-order.

Go ahead and try it out. You'll see that the default order (the order we've seen up 'till now) is actually post-order, or inside-out. The children are generated first, and the resulting subtree is used in the call to the `child-count` rule. The result is

```
(kids 10 ((child) (child) (child) (child) (child)
          (child) (child) (child) (child) (child)))
```

If we don't like this behaviour, we can change the `child-count` rule's order:

```
(child-count *preorder* .
  ,(lambda (tag children)
    '(kids ,(length children) ,children)))
```

This will produce the following result:

```
(kids 1 (children))
```

Wait a minute! That's not what we expected, is it? The `(children)` element isn't transformed anymore! That's because `*preorder*` rules block the transformation process. To obtain truly outside-in behaviour, we need to explicitly call `pre-post-order` in the rule:

```
(child-count *preorder* .
  ,(lambda (tag children)
    (pre-post-order
      '(kids ,(length children) ,children)
      counting-rules))
```

This results in the correct response of

```
(kids 1 ((child) (child) (child) (child) (child)
         (child) (child) (child) (child) (child)))
```

We could've just called `pre-post-order` on the `children`, but the shown pattern is so common that there is a shortcut:

```
(child-count *macro* .
  ,(lambda (tag children)
    '(kids ,(length children) ,children)))
```

This does exactly the same as calling `pre-post-order` on a `*preorder*` rule's result. Be careful not to introduce endless loops this way! If the macro's rule returns an element that is transformed by another rule, it may be possible that there will be no end to the transformations. It is tempting to make everything `*macro*` rules, because very often rules produce new content that also contains tags that need to be rewritten. There are many examples where we need `*macro*`, even if we don't really care about the order of transformation. Here is one:

```
(kids . ,(lambda (tag . contents) '(h2 ,@contents)))
```

The `kids` tag is of course not a valid HTML rule, so we probably want to reduce it further. If we use the `*preorder*` rule, the `kids` node is obviously not reduced to a `h2`. But if we use the original post-order rule (the one without `*preorder*` or `*macro*`), the result doesn't have `pre-post-order` applied to it either. Calling `pre-post-order` on a post-order rule's result is wasteful because it will traverse the whole subtree again. However, if we use `*macro*`, it will traverse the subtree only once. Unfortunately, we'll have to traverse the `children` rule first, and the resulting tag as well, so we can't really evade traversing the tree twice.

```
(child-count .
  ,(lambda (tag children)
    (pre-post-order
      '(kids ,(length children) ,children)
      counting-rules))
```

6 Unescaped content

On certain occasions, you want to enter the raw XML output directly as a string, without having `pre-post-order` escape it for you. For example, we might want to output a HTML document-type:

```
(define broken-page-rules
  '((page *macro* .
    ,(lambda (tag title . rest)
      '((doctype)
        (html (@ (xmlns "http://www.w3.org/1999/xhtml")
          (xml:lang "en") (lang "en"))
          (head
            (link (@ (rel "stylesheet")
              (type "text/css")
              (href "layout.css"))
            (title ,title))
            (body ,@rest))))))
    (doctype .
      ,(lambda (tag)
        (string-append
          "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\" \"
          \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\">"))
        (*text* . ,(lambda (tag str) str))
        (*default* . ,(lambda x x))))))

(define content
  '(page "test"
    (h1 "blah")))

(SRV:send-reply (pre-post-order
  (pre-post-order content broken-page-rules
  universal-conversion-rules))
```

If we try to build a page with these rules, we'll see it goes wrong because the doctype rule is escaped. We can bypass the escaping by extending the `universal-conversion-rules`. These rules are just rules like we've made ourselves. They consist of `*default*` and `*text*` rules that take care of the escaping and translation of lists to tags. We can do it like this:

```
(define page-rules
  '((page .
    ,(lambda (tag title . rest)
      '((doctype)
        (html (@ (xmlns "http://www.w3.org/1999/xhtml")
          (xml:lang "en") (lang "en"))
          (head
```

```

        (link (@ (rel "stylesheet")
                (type "text/css")
                (href "layout.css")))
        (title ,title))
    (body ,@rest))))
(*text* . ,(lambda (tag str) str))
(*default* . ,(lambda x x)))

(define doctype-rules
  '((doctype .
    ,(lambda (tag)
      (string-append
        "<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN\""
        " \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\">")))))

(define content
  '(page "test"
    (h1 "blah")))

(SRV:send-reply (pre-post-order
  (pre-post-order content my-rules)
  (append doctype-rules universal-conversion-rules)))

```

7 A few useful tools

In order to streamline this whole stuff a bit, it's nice to build a few helper procedures. I will now give you a few tools I've found useful when writing my own pages.

7.1 Hassle-free output

;; Requires SRFI-1, for fold

```

(define (sxml-apply-rules content . rules)
  (fold (lambda (rules content)
        (pre-post-order content rules)) content rules))

(define (output-html content . rules)
  (SRV:send-reply (apply sxml-apply-rules content rules)))

```

These two² are very useful, since we often end up nesting a lot of calls to `pre-post-order`, resulting in a big bunch of code just to send the output to the user's browser. That's what `sxml-apply-rules` is for, now we can just call `(sxml-apply-rules document my-rules universal-conversion-rules)`. If

²Chicken users will find these in the spiffy-utils egg.

we don't want the resulting SXML, we can call `output-html` instead of `sxml-apply-rules` and it not only applies `pre-post-order` for each of the rules, but it also sends the output directly to the browser.

7.2 Entities

I have not yet explained how to output HTML entities like `>`. If you simply try to output an entity as a string, you'll see that the `universal-conversion-rules` will recode the `&` to `&`, which means that `>` will look like `&gt;` in the final output. This is definitely not what we want. We'll have to define a rule that we can add to the `universal-conversion-rules`. We'll take the rule from the Chicken scheme system, which already provides one exactly for this reason in its default `universal-conversion-rules` for its `sxml-transforms` egg:

```
(define universal-conversion-rules
  (append
    universal-conversion-rules
    '((& . ,(lambda (tag . elts)
              (map (lambda (elt)
                    (string-append "&" elt ";"))
                  elts))))))
```

Now we can just write a page like this:

```
;; 10 > 1 and 1 < 10
(define document
  '(page "Entities example"
        "10 > 1 and 1 " (& "lt") " 10"))
```

And now it doesn't matter how many rulesets we apply to this, since only the final `universal-conversion-rules` translates.

7.3 Adding classes

Very often, you need to conditionally add a class to an already existing piece of content. It's quite useful to be able to have a procedure that does this.

```
;; Uses sxml-match, SRFI-1 for lset-union
;; and SRFI-13 for string-tokenize and string-join
(define (add-classnames content . new-names)
  ;; If there are no new names, we can simply return the content.
  (if (null? new-names)
      content
      ;; Add the classnames in a clean way, by comparing them
      ;; against the existing tags and only adding them if they're
      ;; not already there.
```

```

(let ((add (lambda (old-names)
             (string-join
              (lset-union string=?
                          new-names
                          (string-tokenize old-names))))))
;; Little hack to force the tag to get matched.
(sxml-match (cons 'tag (cdr content))
             ((tag (@ (class ,old-names) . ,rest) . ,body)
              ‘(,(car content) (@ (class ,(add old-names)) ,@rest) ,@body))
             ((tag (@ . ,rest) . ,body)
              ‘(,(car content) (@ (class ,(add "")) ,@rest) ,@body))
             ((tag . ,body)
              ‘(,(car content) (@ (class ,(add ""))) ,@body))))))

;; Example use:
(add-classnames '(p (@ (class "even")) "blah" "selected")
=>
(p (@ (class "selected even")) "blah")

```

Here, I’ve used the `sxml-match` library by Jim Bender. This is a pattern matching library which doesn’t match s-expressions *literally*, but “knows” about SXML. This means, among other things, that it disregards attribute orderings. That’s why it’s possible to match the `class` in any position even though it’s listed as the first attribute in the pattern. This library is a valuable addition to our toolkit. I’ve hacked around a bit to make it match any tag we feed it by replacing the tag itself in the input to the matcher by a preselected tag called simply `tag`. This is because the first element, like in a macro expression, can’t be variable. I recommend reading the documentation on the SXML-match library if you would like to know more. The library is part of a bigger web framework called “WebIt!”, which also includes a Scheme DSL for generating CSS.

It is certainly possible to exclusively use `sxml-match` for generating your output by macro translation instead of `pre-post-order`. The disadvantage of this approach is that rulesets are not composable like they are with `pre-post-order`. Otherwise, it seems to be pretty much equivalent in functionality. On the other hand, if you don’t like the extra dependency, you could also leave out `sxml-match` and write the `add-classnames` procedure manually, but it’s not going to look pretty.

7.4 Getting child nodes and attributes

Often, you only want to look at the child nodes of an element. SXML can be tricky because it treats attribute nodes as regular child nodes. This means you sometimes want to skip those, if they’re there. On other occasions, you want to be able to assume there are attributes to make your code simpler to follow. These two procedures will help with this:

```
(define (child-nodes contents)
```

```

(sxml-match (cons 'tag (cdr contents))
  ((tag (@ . ,attrs) . ,rest) rest)
  ((tag . ,rest) rest)))

(define (attributes contents)
  (sxml-match (cons 'tag (cdr contents))
    ((tag (@ . ,attrs) . ,rest) (cons '@ attrs))
    ((tag . ,rest) '(@))))

```

7.5 Pretty-printing

The SXML-transforms package also comes with a `pp` procedure for Scheme systems which don't have one natively. This procedure pretty-prints a list structure in a nicely indented way. This is great for debugging your SXML output.

8 Final example

To tie it all together, I'll show a complete example. Suppose we're running a webshop and we would like to have our products listed in a table. This uses definitions from earlier in the text. Those are not reproduced here for brevity.

```

;; Requires SRFI-1 for circular-list and a map that
;; stops after processing the shortest list.

;; Tables, lists etc can be striped visually by
;; adding even/odd class rules in CSS.
(define (stripe . contents)
  '(,(car contents) ,(attributes contents)
    ,@(map (lambda (contents odd?)
             (if odd?
                 contents
                 (add-classnames contents "even"))))
        (child-nodes contents) (circular-list #t #f))))

(define table-rules
  '((table . ,stripe)
    (*default* . ,(lambda x x))
    (*text* . ,(lambda (tag string) string))))

;; These would normally come from a database or file, hence the id field.
(define products
  '((1 "Structure and Interpretation of Computer Programs, 2nd edition"
      "Harold Abelson & Gerald Jay Sussman"
      "978-0262011532")
    (2 "The Art of Computer Programming Volumes 1-3 Boxed Set"

```

```

        "Donald Knuth"
        "978-0201485417")
(3 "The Little Schemer, 4th Edition"
   "Daniel P. Friedman and Matthias Felleisen"
   "978-0262560993")
(4 "The Seasoned Schemer"
   "Daniel P. Friedman and Matthias Felleisen"
   "978-0262561006")
(5 "The Reasoned Schemer"
   "Daniel P. Friedman, William E. Byrd and Oleg Kyselyov"
   "978-0262562140")
(6 "The Scheme Programming Language, 3rd Edition"
   "R. Kent Dybvig"
   "978-0262541480"))))

(define id first)
(define title second)
(define author third)
(define isbn fourth)

(define product-rules
  '((products *macro* .
    ,(lambda (tag)
      '(table
        (tr (th "Title") (th "Author") (th "ISBN"))
          ,@(map (lambda (product)
                  '(tr (td (details-link ,product ,(title product)))
                      (td (details-link ,product ,(author product)))
                      (td (details-link ,product ,(isbn product))))))
            products))))
    (details-link .
      ,(lambda (tag product . contents)
        '(a (@ (href ,(string-append "details.shtml?id="
                                       (number->string (id product))))))
          ,@contents)))
    (*text* . ,(lambda (tag str) str))
    (*default* . ,(lambda x x))))

(define document
  '(page "Product overview"
    (h1 "Products")
    (p "Please click on a product to see its details.")
    (products)))

(output-html document
  product-rules

```

```
table-rules
page-rules
(append doctype-rules universal-conversion-rules))
```

Our `layout.css` can look something like this:

```
.even {
  background-color: #aaff00;
}
```

Now every even row in the table will have a lime background color. Of course, you need to write a `details.xml` for this page to work as it should.

9 More information

If you would like to know more about SXML, visit the [SSAX project homepage](#) and [Oleg Kiselyov's SXML page](#). You can find not only the official specification of SXML here, but also information about other SXML technologies (including how to write XML-to-SXML parsers). Happy Scheming!