Generative XPath

Oleg Parashchenko

Saint-Petersburg State University, Russia
olpa uucode com
http://uucode.com/blog/

XML Prague 2007

Abstract

Generative XPath is an XPath 1.0 implementation, which can be adapted to different hierarchical memory structures and different programming languages. It is based on a small, easy to implement virtual machine, which is already available for different platforms, including plain C, Java and .NET.

1 Introduction

As a consultant in the areas of XML and technical documentation, I often face the task of data transformation. In many cases I can't use convenient tools such as XPath or XSLT (at least, not initially), rather I am limited to the scripting tools of the source platform.

In my projects, the source data is tree-like, and one of the most required of functionalities is navigating over trees. After implementing this functionality several times for different platforms, I noticed I'm writing essentially the same code again and again, with the only difference being the actual programming language and tree data type. There were two problems:

- The code is hard to write, debug and maintain. What is of few characters length in XPath, is several screens of actual code.
- Re-phrasing Greenspun's Tenth Rule of Programming: "Any sufficiently complicated tree navigation library contains an ad hoc informally-specified bug-ridden slow implementation of half of Xpath."

Obviously, it's better to use XPath than custom libraries. But what's if XPath is not available for the platform of choice? Implementing XPath correctly isn't an easy task.

Here is an alternative approach. Generative XPath is an XPath 1.0 processor that can be adapted to different hierarchical memory structures and different programming languages. Customizing Generative XPath to a specific environment is several magnitudes of order easier than implementing XPath from scratch.

The paper is organized as follows. First, I introduce a few use cases for using XPath over hierarchical data. Then the architecture of the Generative XPath approach is explained, followed by a description of the Virtual Machine (VM). The next sections are highly technical and details-oriented, they contain the specification of the interfaces between a program and the VM. The paper continues by highlighting some features of the compiled XPath code. Then we talk about correctness and performance of our system. Finally, we compare Generative XPath with related work and outline further development.

Generative XPath downloads can be found at the XSieve project page:
`http://sourceforge.net/projects/xsieve/`.

2 Use cases

Here is a selection of a few use cases when XPath over something, which is not exactly XML, is possible.

2.1 File system

Many XPath tutorials say that the basic XPath syntax is similar to filesystem addressing. Indeed, in both cases we have relative and absolute paths, and the step separator is slash. Representing the file system as XML is a topic of several projects. For example, one of them, FSX *[FSX]*, was presented by Kaspar Giger and Erik Wilde at the WWW2006 conference.

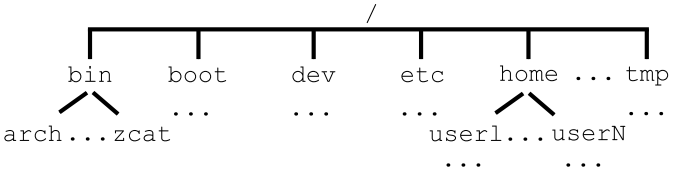

Figure 1: File system as tree

2.2 Compilers and interpreters

Compilers and interpreters deal a lot with abstract syntax trees (ASTs), which appear as the result of parsing expressions. A possible AST for the expression "`1+b+2*3`" is shown on the picture.

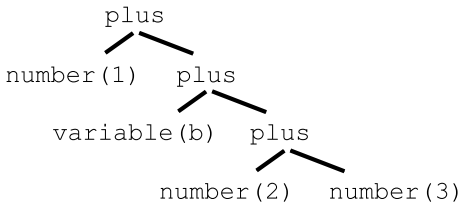

Figure 2: AST for the expression "1+b+2*3"

Suppose that an optimizer wants to calculate expressions at the compilation stage. One of the possible reductions would be to use "mult" operations when operands are numbers. A corresponding XPath over an AST might look so:

```
//mult[count(number)=count(*)]
```

2.3 Text processors

The visual representation of a document introduces an informal structure, with such elements as titles, paragraphs or emphasis.

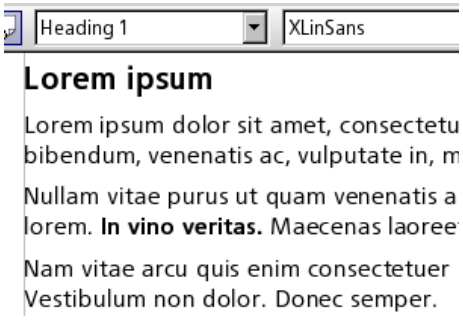

Figure 3: A sample document

The informal structure can be approximated by a tree, in which the nodes are implied by the styles and formatting overrides.

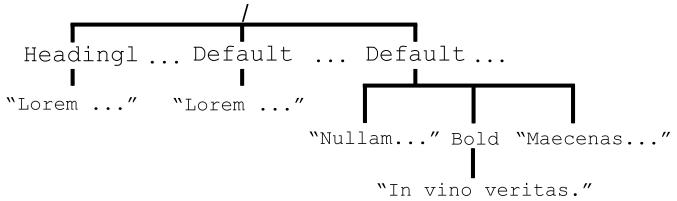

Figure 4: The tree, inferred from the formatting

Now it is easy to get, for example, all the titles of the first level:

```
//Heading1
```

3 Architecture

The Generative XPath approach consists of two main components:

- XPath compiler
- Runtime environment

The XPath compiler transforms XPath expressions to the executable code for the virtual machine (VM). The runtime environment helps an application to execute the compiled code. The runtime environment can be further divided into three logical layers:

- The application layer
- The VM layer
- The customization layer

The application layer is the program which needs an XPath implementation over its tree-like data structures.

The VM layer is:

- The VM itself
- Compiled XPath code
- Runtime support library

The compiled XPath code relies on the runtime support library, which is also written in the VM language.

The customization layer is an intermediate between the application layer and the VM layer, which are completely independent. In particular, the customization layer maps the application-specific trees to a form, suitable for the VM.

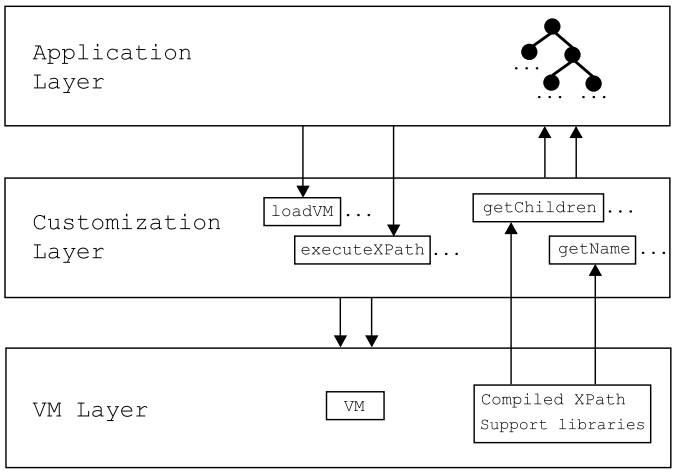

Figure 5: The elements of the Generative XPath architecture

The application layer uses the customization layer to:

- Load and initialize the VM
- Start XPath evaluation and access the result

The VM layer uses the customization layer to:

- Get a collection of nodes by a node (XPath axes)
- Compare nodes in the document order
- Query node properties, such as name or string value

The internal structure of nodes is of no interest to the VM layer, it handles the nodes as "black box" items.

To add XPath support to an application, the developer needs to:

- Find or create a suitable VM implementation
- Implement the customization layer

4 Virtual Machine

Each VM instruction is represented as a list, in which the first element is the name of the instruction, and the remaining elements are the instruction's arguments. Here is an example of a program to calculate factorial:

```
(define (fac n)
  (if (< n 2)
    1
    (* n (fac (- n 1)))))

(fac 1) ; Evaluates to 1
(fac 6) ; Evaluates to 720
```

This code uses the following instructions. "`Define`" creates a new function named "`fac`" which takes one argument "`n`". The conditional instruction "`if`" evaluates the first argument, which is the comparison "`<`", and either returns 1, or continues recursive execution and returns the result of the multiplication.

The program for the VM is actually a program written in the programming language Scheme R5RS *[SCHEME]*. The VM itself is:

- A subset of Scheme R5RS
- A few extension functions
- Functions defined in the customization layer

There are a lot of Scheme implementations to choose from *[IMPLEMENTATIONS]*, some of them are tailored for embedding. For example, C applications can use Guile *[GUILE]*, and Java applications can use SISC *[SISC]*.

As Scheme is a small language, its implementation can be written from scratch with little efforts. A possible approach is explained in Marc Feeley's presentation "The 90 Minute Scheme to C compiler" *[90MINUTE]*. Furthermore, the XPath compiler generates code which doesn't use complex features of the language:

- No continuations, dynamic-wind and related features
- Only a few simple macro definitions are used

Compiled XPath code and the support library rely on the extension functions defined in the following SRFI documents *[SRFI]*:

- SRFI 8: receive: Binding to multiple values
- SRFI 13: String Library
- SRFI 42: Eager Comprehensions

With time, the extension functions will be re-implemented for the VM runtime support library, and the dependencies on SRFIs will be removed.

5 Interacting with the VM

This section explains how the application layer communicates with the customization layer, and the latter with the VM layer. I will give concrete code examples rather than explaining principles only. As a result, this section is highly technical.

The code is written in C for the Scheme implementation "Guile". A completely working program can be found in the Generative XPath distribution in the folder "`examples/c_string`".

5.1 Booleans, numbers, strings

To exchange data with the VM, the application and customization layers need to convert data to a format which the VM can understand (hereinafter referenced with the prefix "vm-"). The exact details depend on the VM implementation.

For example, in Guile, the following functions can be used to convert boolean, number and string values to vm-values:

```
SCM gh_bool2scm(int x)
SCM gh_double2scm(double x)
SCM gh_str02scm(const char *s)
```

Having a vm-value, one gets the value using the following Guile functions and macro:

```
int gh_scm2bool(SCM obj)
double gh_scm2double(SCM obj)
SCM_STRING_CHARS(scm)
```

5.2 Lists

The Scheme list is a recursive data type, consisting of pairs. For each pair, the first element (usually referred to as "`car`") is an item of the list, the second element ("`cdr`") is the tail of the list. Having the `car` and the `cdr`, the list can be re-constructed using the function `cons`. The list is finished when the `cdr` is the special value "empty list".

For example, consider the list "`(1 2 3)`". Its car is "`1`", its `cdr` is the list "`(2 3)`". The list can be re-written as "`(cons 1 (cons 2 (cons 3 '())))`", where "`'()`" is the empty list.

To create or deconstruct vm-lists, use the VM functions for `car`, `cdr` and `cons`. For example, to construct the list "`(1 2 3)`" in Guile, the following code is possible:

```
SCM scm = SCM_EOL; // empty list
for (int i = 3; i; i--) {
  scm = gh_cons(gh_int2scm(i), scm);
}
```

And here is an example of walking over a list. For simplicity, we suppose it consists of integers only:

```
void
print_list_of_integers(SCM list) {
  for (
    SCM scm = list;   // start with the first element
    scm != SCM_EOL;   // walk until the end of list
    scm = gh_cdr(scm) // go to the next element
  ) {
    printf("Integer: %i\n", gh_scm2int(gh_car(scm)));
  }
}
```

5.3 Boxes

There is no such type as "box" is Scheme. It's our extension to represent application's tree nodes. "Box" is a Scheme value, which contains a link to the tree node and possibly additional information, useful for the customization layer.

In other words, box is a form of tree node that can be used in Scheme code.

Creating such objects for Guile is a slightly cumbersome task. It is required to register a new type together with its garbage collection support functions. The code might look like this:

```
//
// Underlying object for boxes
//
typedef struct _vmbox {
  my_node *node; // pointer to the tree node
  ...            // more data, if required
} vmbox;

//
// Initialization of a new type, attaching the
// garbage collection functions and the destructor.
//
scm_t_bits vmbox_tag;
...

//
// Creating a box
//
SCM
make_box(my_node *node, ...extra...) {
  SCM smob;
  vmbox *box = (vmbox*)scm_gc_malloc(
                        sizeof(vmbox), "box");
  box->node  = node;
  ...extra...
  SCM_NEWSMOB(smob, vmbox_tag, box);
  return smob;
}

//
// Unboxing a node
//
my_node *
unbox(SCM vmbox) {
  scm_assert_smob_type(vmbox_tag, vmbox);
  vmbox *box = (vmbox*)SCM_SMOB_DATA(scm_box);
  return box->node;
}
```

Details of creating and managing custom types in Guile are described in the Guile manual *[GUILEMAN]* in the section "Defining New Types (Smobs)".

5.4 Boxes-2

The application layer and the VM layer don't care how boxes are created and managed. These details are left to the customization layer. However, here are few guidelines.

The only case when the application layer can create a box from a node is when the node is the root of a tree. All other boxes are created only inside the customization layer.

The only operations on boxes, available to the application layer, are:

- unboxing (extracting the node from a box), and
- applying an XPath.

5.5 Initializing the VM layer

The VM layer is initialized with help of the customization layer. The initialization steps are as follows:

- Initialize the VM
- Set up the environment for the VM, such as include paths for libraries
- Load VM extensions, such as SRFI extensions
- Create the box type and register the interface functions
- Load the Generative XPath runtime library

5.6 Loading XPath code

The XPath compiler transforms an XPath expression into Scheme code with the free variables "`c`", "`k`" and "`n`" (the context node, position and size, respectively). Before loading the code, an application should bind these variables, otherwise loading might fail.

The details are not shown here. Instead, an alternate approach is suggested:

- Convert each XPath code to a procedure
- Load all procedures at once
- Locate the procedures in the Scheme environment

Converting XPath code is a simple task, the code should be wrapped this way:

```
(define (xpathNNN c k n)
  ... original code goes here ...
)
```

The code fragment above defines a procedure with the name "xpathNNN" (it's the responsibility of the application to assign unique names) and three context parameters.

Supposing all such definitions are stored in the file "vmcode.scm", they can all be loaded at once:

```
gh_eval_file("vmcode.scm");
```

Finally, here is how to locate the loaded procedures:

```
SCM  xpathNNN = gh_lookup("xpathNNN");
if (xpathNNN == SCM_UNDEFINED) {
  puts("XPath procedure 'xpathNNN' isn't found");
}
```

5.7 Executing loaded XPath

Suppose the variable "xpath" contains a loaded compiled XPath code, wrapped to a procedure as described in the previous section. Before calling the procedure, its arguments should be prepared.

The first argument is a context node in its box form. The application layer gets the box as the result of boxing the root node or as the result of executing another XPath expression.

The second and the third arguments are the context position and context size, respectively. Unless the application uses XPath expressions with the functions "position()" and "last()" at the top level (outside any predicate), these arguments are ignored. Otherwise, they should be vm-integers, and the application should supply the correct values.

Guile example:

```
//
// xpathNNN is an XPath procedure
// "box" is the context node
// We don't care about context position and size
//
ret = gh_call3(xpath, box,
           SCM_UNSPECIFIED, SCM_UNSPECIFIED);
```

5.8 Interpreting the result

Generative XPath always returns the result as a list. The actual type of the result is interpreted as follows:

- If the list is empty, the result is the empty node set.
- If the type of the first element of the list is boolean, number or string, then this element is the result.
- Otherwise, the result is the node set.

Example of interpreting the result in Guile:

```
if (SCM_EOL == ret) {
  puts("empty node set");
} else {
  SCM node, val = gh_car(ret);
  if (SCM_BOOLP(val)) {
    printf("boolean, %i\n", gh_scm2bool(val));
  } else if (SCM_NUMBERP(val)) {
    printf("number, %f\n", gh_scm2double(val));
  } else if (SCM_STRINGP(val)) {
    printf("string, %s\n", SCM_STRING_CHARS(val));
  } else {
    printf("nodeset:\n");
    for(; SCM_EOL != ret; ret = gh_cdr(ret)) {
      val  = gh_car(ret);
      node = unbox(val);
      print_node(node);
    }
  }
}
```

5.9 Interface to the tree

This section described the functions which return information about the application's trees. These functions are the part of the customization layer.

```
box gx-ffi:root(box c)
```

Input is the context node "c", the function should return the root node of the tree. Note that the root node and the highest-level element are two different things.

```
list gx-ffi:axis-child(
    box           c,
    symbol        node-test,
    string/symbol namespace-uri,
    string/symbol local-name)
```

And twelve functions with the same input and output parameters: "gx-ffi:axis-descendant", "gx-ffi:axis-parent", "gx-ffi:axis-ancestor", "gx-ffi:axis-following-sibling", "gx-ffi:axis-preceding-sibling", "gx-ffi:axis-following", "gx-ffi:axis-preceding", "gx-ffi:axis-attribute", "gx-ffi:axis-namespace", "gx-ffi:axis-self", "gx-ffi:axis-descendant-or-self", "gx-ffi:axis-ancestor-or-self".

These functions should return the corresponding XPath axes for the node "c". The nodes in axes are sorted in the document order, except for the four axes in which the nodes are sorted in the reverse document order: "ancestor", "ancestor-or-self", "preceding", "preceding-sibling".

The symbol "node-test" defines the filter for the node type. The range of the values is: "*", "text", "comment", "processing-instruction", "node". The semantics are identical to those defined in the XPath specification. Note that "*" doesn't mean "all the nodes", it means "all the nodes of the axis' principal type".

The strings "namespace-uri" and "local-name" are the filters for the names of the nodes. Alternatively, if these parameters are the symbol "*", then no filtering by name is required.

```
number/boolean gx-ffi:<=>(box n1, box n2)
```

The function compares the nodes in the document order. It returns:

- `-1`, if the node "`n1`" comes before the node "`n2`",
- `0`, if "`n1`" and "`n2`" are the same nodes,
- `1`, if "`n1`" comes after "`n2`",
- `#f`, in case of error.

The function can return #f if the nodes are from different documents. But it also can return -1 or 1, as long as the result is consistent for all the nodes of these documents.

```
string gx-ffi:string(box n)
string gx-ffi:namespace-uri(box n)
string gx-ffi:local-name(box n)
string gx-ffi:name(box n)
```

These are the counterparts to the XPath functions "`string()`", "`namespace-uri()`", "`local-name()`" and "`name()`". The special cases such as *no parameters* or *a nodeset as the parameter* are handled by Generative XPath itself. These functions have only one node as the parameter.

```
boolean gx-ffi:lang(box c, string lang)
node/boolean gx-ffi:id(box c, string id)
```

These are the simplified counterparts to the XPath functions "`lang()`" and "`id()`". The variable "`c`" is the context node. The function "`gx-ffi:id()`" should return either a node, or `#f` (false).

6 XPath compiler

The description of the XPath compiler consist of two parts:

- The tools
- Outline of the internals of the compiler.

6.1 Tools

The XPath compiler is a standalone command-line program "`ast_to_code.scm`" (wrapped by the shell script "`run.sh`"). The program gets an XPath expression as a command line argument and dumps the compiled XPath code to the standard output.

```
$ ./run.sh '//a/b'
```

The section "Loading XPath code" suggests to wrap each compiled code in a procedure and store the result in a file. The program "`bulk.scm`" performs exactly this task. Edit the file, adding the required XPath expressions, and run it.

There is also an improved version of the program, "`bulk_escaped.scm`", which allows for compilation of XPath expressions with invalid XML names by URI-escaping them. For example, instead of writing

```
/*[name()='C:']/*[name()='Program Files']
```

we can use a more nice expression

```
/C%3A/Program%20Files
```

The compiler is written in Scheme. If an application embeds the VM as a full-featured Scheme implementation, then the application can compile XPath expression in runtime using the library function `gx:xpath-to-code-full`.

6.2 Compiler internals

The compiler parses and rewrites XPath expressions using the libraries of the project ssax-sxml *[SXML]*. In particular, the first step is to convert XPath to an abstract syntax tree (AST) using the function "`txp:sxpath->ast`". The AST tree is rewritten, bottom-up, to code using SXSLT, a Scheme counterpart of XSLT.

The code is generated as straightforwardly as possible, with no attempts at optimization. It is supposed that the generated code is later processed by an optimizer. Here is an example of the code generated for "`1+2`":

```
(gx:unit
  (+ (gx:number (gx:sidoaed (gx:unit 1)))
     (gx:number (gx:sidoaed (gx:unit 2))))))
```

In our approach, each XPath step should return a sequence, therefore we wrap atomic values to lists using the function "`gx:unit`". When processing "`+`", the compiler doesn't look down the AST tree in search for optimization, but generates the common code. First, the arguments are sorted in the document order ("sidoaed" stands for "sort in document order and eliminate duplicates"), and than converted to numbers. The code is suboptimal, but, fortunately, easy to optimize.

The majority of the XPath core functions are implemented in the runtime support library. For the remainder, the library takes care of the special cases, defined in the XPath specification, and calls the customization layer only if required. For example, here is the code of the function "`gx:string`":

```
(define (gx:string nset)
  (if (null? nset)
    ""
    (let ((val (car nset))) (cond
```

```
((boolean? val) (if val "true" "false"))
((number?  val) (cond
          ... NaN/infinity/integer cases ...
          (else (number->string val))))
((string?  val) val)
(else           (gx-ffi:string val))))))
```

It checks, in order, if the argument is the empty node set, or boolean, number or string value, and acts accordingly. Otherwise, the argument is a node set, sorted in the document order, and the result is the string value of the first node, as returned by the customization layer.

Compilation of XPath step sequences is non-trivial. The syntactic form for it is called "list comprehension". The expression "`step1/step2`" is compiled to:

```
(list-ec
    (:list c   ... step1 code ...)
    (:list c2  ... step2 code ...)
  c2))
```

The literal interpretation is as expected: execute `step1`, bind the variable `c` to each node of the result, execute `step2` and bind the variable `c2` to each node of the result. The whole result of `list-ec` is the list of the values of `c2`.

A similar construction is used to represent filtering. The only added complexity is preprocessing the input list and annotating the nodes with the context position and size, and unpacking the annotations before evaluating the predicate.

The main feature of the generated code (and the support library) is that it is recursion-free. Only higher-order constructions, such as list comprehension and morphisms, are used. As result, it is possible to substitute all the function calls by the bodies of the functions and get one big code block. I believe it allows to perform aggressive optimization of the generated code, and it's the topic of further investigations.

7 Compliance and performance

Correct implementation of the XPath 1.0 standard is the main constraint for the Generative XPath project. To reach this goal, I use:

- A set of unit tests
- Testing with real world stylesheets

For the latter, I use XSieve *[XSIEVE]*, a mix of XSLT and Scheme. Initially, many XSLT constructions are converted to the XSieve form. For example,

```
<xsl:apply-templates select="@*|node()" >
```

becomes

```
<s:scheme>
  (x:apply-templates (x:eval "@*|node()"))
</s:scheme>
```

Secondly, I extract all the "`x:eval`" expressions and replace them with the corresponding compiled code. As a result, most of the XPath selects are performed by Generative XPath.

Now we can run the original stylesheet, then the Generative XPath version of the stylesheet, and compare the results. They should be the same. I executed this test for DocBook stylesheets, and the test was passed successfully. Therefore, with a high degree of confidence, Generative XPath is a compliant implementation of XPath.

Performance is still a topic of further work. The generated code is not optimized for speed, instead, it is in a form suitable for analysis by an optimizer, which is under construction.

In the worst case, in an unfair setup and measurement, Generative XPath is 30 times slower than a libxml2 implementation of XPath. However, I expect that speed is comparable in the case of fair conditions, and Generative XPath might be faster when the optimizer is implemented.

8 Related work

The majority of XSLT/XQuery processors somehow allow custom trees in theirs XPath engines, but this is mostly a side-effect, not intended functionality. I'm aware of a few projects, in which virtualization of XML is the main selling point:

- Jaxen *[JAXEN]* (Java),
- JXPath *[JXPATH]* (Java) from Apache Software Foundation,
- XML Virtual Garden *[XMLVG]* (Java) from IBM,
- `IXPathNavigable` interface *[IXPATH]* (.NET) from Microsoft,
- XLinq *[XLINQ]* (.NET) from Microsoft.

The main limitation of these projects is that they are bound to the platform of choice. For example, we can't use Jaxen if we develop in .NET. Worse, we can't use any of the mentioned tools if we are programming in plain C.

Unlike the above projects, Generative XPath is highly portable as it is based on a small, easy to implement virtual machine.

9 Conclusion and further work

Generative XPath is an XPath 1.0 processor, which can be adapted to different hierarchical memory structures and different programming languages. Generative XPath consists of the compiler and the runtime environment. The

latter is based on a simple virtual machine, which is a subset of the programming language Scheme.

In this paper, we've explained why XPath over arbitrary tree-like structures is useful at all, described the architecture, interfaces, tools and internals of Generative XPath, and given links to the related developments.

The next step for the Generative XPath project is the development of an optimizer. The corresponding announcements and downloads will be available from the XSieve project page:
http://sourceforge.net/projects/xsieve/.

10 References

[90MINUTE] Marc Feeley: The 90 Minute Scheme to C compiler. http://www.iro.umontreal.ca/~boucherd/mslug/meetings/20041020/minutes-en.html

[FSX] K Giger, E Wilde: XPath filename expansion in a Unix shell, in Proceedings of the 15th international conference on World Wide Web, 2006. http://www2006.org/programme/files/xhtml/p95/pp095-wilde.html

[GUILE] Free Software Foundation, Inc.: Guile (About Guile).
http://www.gnu.org/software/guile/guile.html

[GUILEMAN] Free Software Foundation: GNU Guile Manual.
http://www.gnu.org/software/guile/manual/

[IMPLEMENTATIONS] Schemers.org: Implementations.
http://schemers.org/Implementations/

[IXPATH] Microsoft Corporation. XPathNavigator in the .NET Framework. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconXPathNavigatorOverDifferentStores.asp

[JAXEN] Apache Software Foundation: JXPath - JXPath Home. http://jakarta.apache.org/commons/jxpath/index.html

[JXPATH] jaxen: universal Java XPath engine - jaxen.
http://jaxen.org/

[SCHEME] Kelsey, R., Clinger, W., Rees, J. (eds.): Revised5 Report on the Algorithmic Language Scheme. Higher-Order and Symbolic Computation, Vol. 11, No. 1, August, 1998. http://www.brics.dk/~hosc/11-1/

[SISC] Scott G. Miller: Second Interpreter of Scheme Code. http://sisc-scheme.org/

[SRFI] The SRFI Editors: Scheme Requests for Implementation. http://srfi.schemers.org/

[SXML] Kiselyov, O.: SXML Specification.
http://okmij.org/ftp/Scheme/xml.html#SXML-spec

[XSIEVE] Paraschenko, O.: XSieve book.
http://xsieve.sourceforge.net/

[XLINQ] Erik Meijer and Brian Beckman: XLINQ: XML Programming Refactored (The Return Of The Monoids), in Proceedings of XML 2005, November 2005.
http://research.microsoft.com/~emeijer/Papers/XMLRefactored.html

[XMLVG] IBM alphaWorks: Virtual XML Garden.
http://www.alphaworks.ibm.com/tech/virtualxml