# A better XML parser through functional programming

oleg@pobox.com oleg@acm.org

Presented at the 4th International Symposium
*Practical Aspects of Declarative Languages*
PADL'02, January 19-20 2002, Portland, OR, USA.

The corresponding paper is published in LNCS 2257, Shriram Krishnamurthi and C.R. Ramakrishnan, Eds. Springer-Verlag Berlin Heidelberg, 2002.

Is XML simple?

- Attribute normalization rules
- Parsed and parameter entities
- White space
- XML Namespace
- Checking well-formedness and validation constraints: element content model, attribute value content model, ID uniqueness, etc.

Before I get to the meat of the talk, I have to dispel a *persistent* myth that XML is trivial and XML parsing is trivial. Emphatically there is more to XML that the grammar presented in the XML Recommendation. There are attribute normalization rules, well-formedness constraints, let alone validation constraints. XML Namespaces add another layer of complexity.

Note that the current version of XML Recommendation is called "Version 1.0, Second Edition". The language itself hasn't changed. However, it was recognized that the Recommendation had to be clarified. This alone is an indicator of complexity **{**and of the fact that XML isn't specified formally). The treatment of < and of line terminators required several clarifications. It turns out that some productions and rules were having unintended effect.**}** As one example, errata E62 changed the interpretation of white spaces in NAMES and NMTOKENS productions; it was later rescinded by errata E108. The latter was rescinded by E20 errata of the second edition. It's hard to believe that white space would cause so much controversy.

Attribute value normalization rules

(section 3.3.3 of the XML Recommendation)

- replace TAB, CR, LF and CRLF with a single space

- expand character references, e.g., `
`

- expand internal entity references, e.g., `<`
  - normalize the expansion text
  - check for no character '<'
  - treat quotes as regular characters
  - expand nested character and internal entities
  - check a non-recursion constraint: "A parsed entity must not contain a recursive reference to itself, either directly or indirectly"

One example of XML complexity is attribute value normalization rules **{**(Section 3.3.3 of the XML Recommendation)**}**. To obtain the value of an attribute, it is ***not*** enough to merely read text between two single or double quotes. A parser shall replace TAB, CR, LF characters and a CRLF character combination with a single space. A parser shall expand character references, e.g., 
 A parser shall expand internal entity references, e.g., <. The expansion text of an internal entity shall also be normalized. A parser shall check that the replacement text does not contain a character '<' verbatim. A parser shall treat quotes that may appear in the replacement text as regular characters of no special meaning. The replacement text may contain references to other character and internal entities, which must be expanded and normalized in turn. Furthermore, the parser must keep in mind a non-recursion constraint: "A parsed entity must not contain a recursive reference to itself, either directly or indirectly". Even a *non*-validating XML parser *must* follow all these rules. <I'm out of breath just reading all these rules>.

SAX vs. DOM

- Document Object Model (DOM):
  Parsing is XML $\rightarrow$ AST

- Simple API for XML (SAX):
  Parsing as a sequence of events

SAX is

- more general: can produce DOM and other data structures

- memory efficient, especially for huge documents

- faster

- more difficult to use

There are two styles of parsing XML. The traditional view is a transformation of a source document into an abstract syntax tree. This is called a Document Object Model (DOM) parsing. The DOM approach is called for for applications that repeatedly traverse and search the abstract syntax tree of a document. Other applications however scan through the document tree entirely, and only once. Such applications can potentially process an XML document as they read it. Loading the whole document into memory as an abstract syntax tree is then inefficient both in terms of time and memory. Such applications can benefit from a lower-level, event-based model of XML parsing called a Simple Application Programming Interface for XML (SAX). A SAX parser applies user-defined actions to elements, character data and other XML entities as they are identified. The actions can transform received elements and character data on the fly, or can incorporate them into custom data structures, including the DOM tree. Therefore, a SAX parser can always act as a DOM parser. The converse is not true.

Although the event-based XML parsing model is more general, more memory efficient and faster, SAX parsers are regarded as difficult to use. We will demonstrate that later.

SSAX parser

- stream-oriented SAX XML parser
- used in industrial applications
- supports XML Namespaces, parsed entities, attribute value normalization, ...
- written in a pure referentially-transparent subset of Scheme
- has a *better API*
  - minimizes the amount of application-specific state that has to be shared among user-supplied event handlers
  - Event handlers are simpler, and simpler to reason about

http://ssax.SourceForge.net/

The goal of this talk is to show by construction that it is possible to implement an efficient, compliant, stream-oriented XML parser with a *convenient* user interface. Furthermore, Functional programming was indeed of great help in designing and implementing such a parser.

The parser, called SSAX, is a compliant SAX XML parser that is being used in several industrial applications. SSAX is not a toy parser: it fully supports XML Namespaces, character, internal and external parsed entities, attribute value normalization, etc. At the same time, SSAX minimizes the amount of application-specific state that has to be shared among event handlers. SSAX is written in a pure-referentially-transparent subset of Scheme. The event handlers are referentially transparent and simpler – which makes them easier for a programmer to write and to reason about. The *better* user application interface for the event-driven XML parsing is in itself a contribution of the paper. We will show that this interface is not an accident but the outcome of a correctly chosen control abstraction, which captures the pattern of depth-first traversal of trees.

## Outline

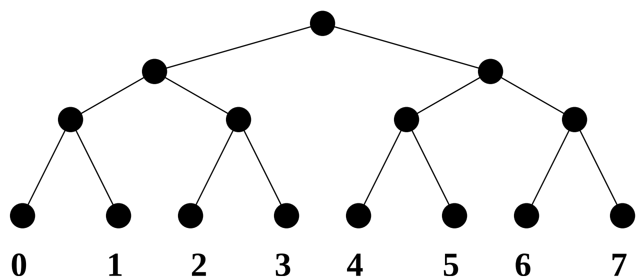
- Complexity of XML
- SAX vs. DOM
- XML parsing as tree traversal
- SSAX and other XML parsers
  - FXP: compare API
  - Expat: compare performance
  - CL-XML (Common Lisp), XISO (Scheme), Tony (OCaml), HaXml ( Haskell), XMerL (Erlang).

The rest of the talk will have two parts: a theoretical part that derives the SSAX API and a practical part that compares SSAX with other functional-style parsers and with the Expat.

Of these functional-style parsers, only XMerL supports the XML Namespaces (but not fully). All but two functional-style XML parsers barely comply even with a half of the XML Recommendation. With the exception of FXP and to some extent XMerL, the existing functional-style parsers cannot process XML documents in a stream-wise fashion. The application interface of the only one functional, full-featured, stream-oriented parser FXP mirrors the API of the reference XML parser Expat. Expat is notorious for its difficult and error-prone application interface.

Let us indeed look into the parser's interface.

# XML document as a tree




```
<node><node><node><leaf>0</leaf><leaf>1</leaf>
            </node>
            <node><leaf>2</leaf><leaf>3</leaf>
      </node></node>
      <node><node><leaf>4</leaf><leaf>5</leaf>
      </node>
            <node><leaf>6</leaf><leaf>7</leaf>
</node></node></node>
```

Before I dive into theory I'd like to make a remark. Speaking from personal experience, it's common to hack something together and *then* bring up theory to "justify" it. Especially for such a noble purpose as submitting a paper. **{**This is a common strategy in many areas of science, e.g., theoretical chemistry, which has a peculiar property that it can explain everything but predict precious little.**}** This is **NOT** what happened in the present paper. The current version of the SSAX parser started with an insight that XML parsing is a tree traversal. I was also influenced by Ralf Hinze's talk at ICFP'00, which led to the idea that an XML document is a composite term, and the parser is its interpreter. The insight and the derivation of the API were first written in the comments. The code was written afterwards, following the comments. I can offer the RCS records as a proof.

Let's consider a sample XML document. We will return to it in the benchmarking section of the talk. We note that an XML document with familiar angular brackets is a concrete representation of a tree laid out in a depth-first order. Elements, processing instructions, and character data are the nodes of such a tree. Character data are always the leaves. **{**Elements in this document do not have attributes; if they had, the attributes would be collections of named values attached to the corresponding element nodes. Since element nodes can be non-terminal nodes, the moments the traversal enters and leaves an element node must be specifically marked, respectively as the start and the end tags.**}**

XML processing is then traversing this tree and executing certain operations on the nodes as we visit them. The parser reads the document sequentially, *this way*. This is precisely the depth-first traversal order of a tree. **{**Operations on nodes include building and linking DOM objects, searching or matching with a specific element, or storing data in a database.**}** XML processing is a pre-post-order, down-and-up traversal as it invokes user actions when the traversal process enters a node and again when the process has visited all child branches and is about to leave the node.

Can we separate the task of tree traversal and recursion from the task of transformation of a node and a state? The benefits of encapsulating common patterns of computation as higher-order operators instead of using recursion directly are well-known. For lists, the common pattern of traversal is captured by the familiar `fold` operators. If we can find such an operator for XML tree traversals – if we can *abstract* the pattern of such traversals – then the operator becomes the XML parser and its parameters are call-backs. We can *derive* the XML parser and its application interface.

Folding over a tree

```
data Tree = Leaf String | Ref String | Nd [Tree]
```

```
foldt::  (String -> a) -> ([a] -> a) -> Tree ->a

foldt f g (Leaf str) = f str

foldt f g (Nd kids) = g (map (foldt f g) kids)
```

```
str_value = concat .  foldt (:[]) concat
```

O(n logn) complexity in time and garbage

This is exactly the way the SSAX parser was derived. **{**We need to find an operator that encapsulates the pattern of the depth-first traversal of trees.**}** XML document tree (similar to the one on the previous slide) has the following structure. Let's forget about *this term* for a moment and concentrate on trees whose leaves are strings.

We start by checking out a tree fold **{**– a generalization of foldr for trees, or an instantiation of a general categorical fold for trees**}**. It takes two actions: one to execute on leaves of the tree and the other on the processed children.

As an example, suppose we want to concatenate strings attached to all leaves, in their depth-first traversal order. In XML-speak, this is the problem of computing a `string-value` for the root node of the tree. We can write the solution using the tree fold: the leaf action turns a string into a one-element list; the `g` action concatenates such lists; finally; we concatenate all the strings into one big string. This code has two obvious drawbacks: it wastes both time and memory at n*logn complexity. The best solution is to build a list of strings in the reverse order – with the reversal and concatenation at the very end. But that solution cannot be expressed as foldt, at least not directly. The biggest problem is this `map` function. It means that children of a node are processed in an indeterminate order, in parallel, so to speak. If we want to build a list of leaf strings in reverse order, we want to accumulate strings *one after* *another*, that is, strictly *sequentially*. In an example of string_value problem, we only lose in efficiency. However, some problems cannot be expressed with the tree fold at all. Imagine that we want to compute the MD5 digest of the tree. Unlike the list append, the MD5 digest function is not associative. Many other stream processing functions we want to do with an XML document are not associative. That is, actions at branches are dependent on the history of the traversal and cannot be simply *mapped* to children nodes.

**{**There is a way around the predicament: higher-order or continuation-passing folds. Essentially such a fold replaces this tree of data with a tree of actions (closures), which are then executed. We may gain good time complexity but we certainly lose in terms of memory. In languages like Scheme closures are relatively expensive. Furthermore, this method does not apply if the entire document (tree) is too big to fit into available memory.**}**

Thus, Fold over list is single threaded while fold over a tree is multi-threaded (all branches are folded independently). What we need is a single-threaded folding over the tree. So, we make one.

## Extended tree fold

```
foldts fdown fup fhere seed (Leaf str) =
   fhere seed str
foldts fdown fup fhere seed (Nd kids) =
   fup seed $
   foldl (foldts fdown fup fhere) (fdown seed)
         kids

str_value = concat . reverse . (str_value' [])
  where
    str_value' = foldts id (\_->id) (flip (:))

tree_digest = md5Final .(foldts fd fu fh md5Init)
  where fh ctx str = md5Update "/leaf" $
                md5Update str $ md5Update "leaf" ctx
        fd ctx = md5Update "node" ctx
        fu _ ctx = md5Update "/node" ctx
```

**{**To make "mapping" of an accumulating, stateful function to a tree efficient, we introduce a more general control operator, foldts.**}** A user instantiates foldts with _three_ actions; for comparison, the list fold requires only one action and the tree foldt needs two. The three actions of foldts are threaded via a 'seed' parameter, which maintains the local state. An action accepts a seed as one of its arguments and returns a new seed as the result. Action fhere is applied to a leaf of the tree. Action fdown is invoked when a non-leaf node is just entered. The fdown action has to generate a seed to pass to the first visited child of the node. Action fup is invoked after all children of a node have been seen. The action is a function of **two** seeds: the parent seed, and the result of visiting all child branches. **{**The fup action is to produce a seed that is taken to be the state of the traversal after the process leaves the current branch.**}** The two previously mentioned examples – computation of a string value and of a digest for a tree – can easily be written with foldts. In this example, the seed is the list of leaf values accumulated in the reverse order. The fhere action prepends the value of the visited leaf to the list. Actions fdown and fup are trivial: they merely propagate the seed. The computation of the tree digest is no more complex. The seed is the MD5 context. The fdown and fup actions mark the fact of entering and exiting a non-leaf node. This example clearly demonstrates that consuming node values and updating the local state are separated from the task of traversing the tree and recurring into its branches.

Incidentally, foldts can also be used for breadth-first traversal and accumulation. <see a backup slide>

Benchmarks and comparisons: API

FXP

- stream-oriented validating SAX XML parser
- purely-functional equivalent of Expat
- written in SML
- relies on parameterized modules for customization
- event-based interface with referentially-transparent "hooks"
- author: Andreas Neumann

http://www.Informatik.Uni-Trier.DE/~aberlea/Fxp/

From the theoretical depths we now get back to the surface, to see if the theoretical derivation of the parser engine really gave us anything. We will compare SSAX with the other simple API parser, namely FXP. FXP is a purely functional, validating XML parser "shell" written in SML. {Both SSAX and FXP invoke user-supplied handlers (called "hooks" in FXP) at "interesting" moments during XML parsing. The hooks receive an application state parameter and must return a possibly new state.}

It can be said that FXP is a purely functional equivalent of the reference XML parser, Expat. The differences in API between SSAX and FXP is the differences between SSAX and any other simple API XML parsers. Because FXP is as purely referentially-transparent as SSAX, the comparison of both parser's interfaces is more meaningful and revealing.

Simple DOM parser in FXP

```
structure TreeData =
struct
  exception IllFormed
  type Tag = int * HookData.AttSpecList
  datatype Tree = TEXT of UniChar.Vector
                | ELEM of Tag * Content
  withtype Content = Tree list
end
```

A sample FXP application discussed in the FXP API documentation is a good example to illustrate that difference. The application converts an XML document to an abstract syntax *tree form*, which is not unlike the Tree datatype we saw previously. A SSAX distribution includes a similar function. It is instructive to compare event handlers of the two applications.

DOM parser in FXP (hooks)

```
structure TreeHooks =
struct
  open IgnoreHooks TreeData UniChar
  type AppData = Content * (Tag * Content) list
  val appStart = (nil,nil)

  fun hookStartTag ((content,stack),
                    (_,elem,atts,_,empty)) =
   if empty then
      (ELEM ((elem,atts),nil) :: content,stack)
   else (nil,((elem,atts),content) :: stack)

  fun hookEndTag ((_,nil),_) = raise IllFormed
    | hookEndTag
      ((content,(tag,content') :: stack),_) =
      (ELEM (tag,rev content) :: content',stack)

  fun hookData ((content,stack),(_,vec,_)) =
      (TEXT vec :: content,stack)
...
end
```

In the FXP application, the application data – or the seed – represents the partial document tree constructed so far. The seed has two components – a *stack* and the *content*. At any point the stack holds all currently open start-tags along with a list of their left siblings. The content component accumulates the children of the current element that are known so far. In the *initial state*, both components are empty. *Character data event handlers* add the identified character data to the content of the current element. The hook for a *start-tag* pushes that tag along with its parent's content onto the stack. Function *hookEndTag* reverses the accumulated content of the current element, pops the tag of the current element off the stack and constructs the branch. The branch is then prepended to the content of the parent element.

Maintenance of FXP's stack was split across *two* separate hooks: handlers for the start and the end tags. There may be a lot of code in-between. It's not trivial to guarantee that the stack is manipulated consistently.

The function that pops data off the stack may of course find the stack empty. The programmer therefore must anticipate that snafu and do something intelligent – raise an error, for example.

Simple DOM parser in SSAX

```
(define (simple-XML->SXML port)


   (reverse
       ((SSAX:make-parser
          NEW-LEVEL-SEED
          (lambda (elem-gi attributes namespaces
                   expected-content seed)  '())

          FINISH-ELEMENT
          (lambda (elem-gi attributes namespaces
                   parent-seed seed)
            (cons (cons elem-gi (reverse seed))
                  parent-seed))

          CHAR-DATA-HANDLER
          (lambda (string1 string2 seed)
            (if (string-null? string2)
                (cons string1 seed)
                (cons* string2 string1 seed)))
        )
        port '())))
```

The SSAX application code (here it is, on this slide) does correspond to the FXP application's description to a certain extent. However, `simple-XML->SXML` is notably simpler. Whereas FXP's application state is comprised of a stack and the content, t*his function*'s state is a regular list. The list contains the preceding siblings of the current entity, in reverse document order. In contrast to FXP, the handlers of `simple-XML->SXML` are relieved of any stack maintenance responsibility. This function does not have any stack. The SSAX handlers (unlike those of FXP) do not need to check for the stack underflow and do not need to raise an `IllFormed` exception. As you see the handlers of `simple-XML->SXML` hardly do anything at all. The new-level-seed handler is particularly trivial; finish-element is not more complex either. We see now the advantage of passing the finish-element handler two seeds: one seed from the branch traversal, and the parent seed. Anyway, the simpler the handlers are, the easier it is to write them and to reason about them.

We should point out that not only `simple-XML->SXML` lacks a stack, the SSAX parsing engine itself does not have an explicit stack of currently open XML elements. The traversal stack is implicit in activation frames of a recursive handle-start-tag procedure of the engine. If there is no explicit stack, there can be no stack underflow errors. Thus the comparison between FXP and SSAX indicates that the SSAX framework provides a higher level of abstraction for event-based XML parsing. This is the direct consequence of building SSAX around the foldtsl tree traversal combinator.

Benchmarks and comparisons: performance

Expat

- "stream-oriented" SAX XML parser
- imperative
- written in C
- event-based interface with imperative callbacks
- *the* XML parser. The fastest XML parser
- author: James Clark

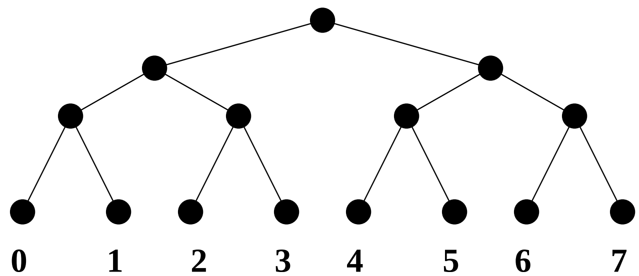
No discussion of XML parsing can avoid Expat, which is *the* reference XML parser, written in C by James Clark. Expat is a stream-oriented, event-based parser. **{**As the user passes it chunks of the input XML document, Expat identifies elements, character data or other entities and invokes the appropriate handler (if the user has registered one). The size of the chunks is controlled by the user; chunks can range from one byte to the whole XML document.**}**

## Benchmark SSAX code

```
(define (remove-markup xml-port)
  (let ((result
          ((SSAX:make-parser
             NEW-LEVEL-SEED
             (lambda (elem-gi attrs namespaces
                      expected-content seed)
               seed)

             FINISH-ELEMENT
             (lambda (elem-gi attrs namespaces
                      parent-seed seed) seed)

             CHAR-DATA-HANDLER
             (lambda (str1 str2 seed)
               (let* ((seed (cons str1 seed)))
                 (if (string-null? str2) seed
                     (cons str2 seed))))
           )
           xml-port '())))
     (string-concatenate-reverse result)
    ))
```

To compare SSAX and Expat in performance, and to check the complexity of SSAX, we chose the following benchmark. The benchmark does untagging. This is a common "XML to text" translation that removes all markup from a well-formed XML document. We have already seen this example.

XML document as a tree



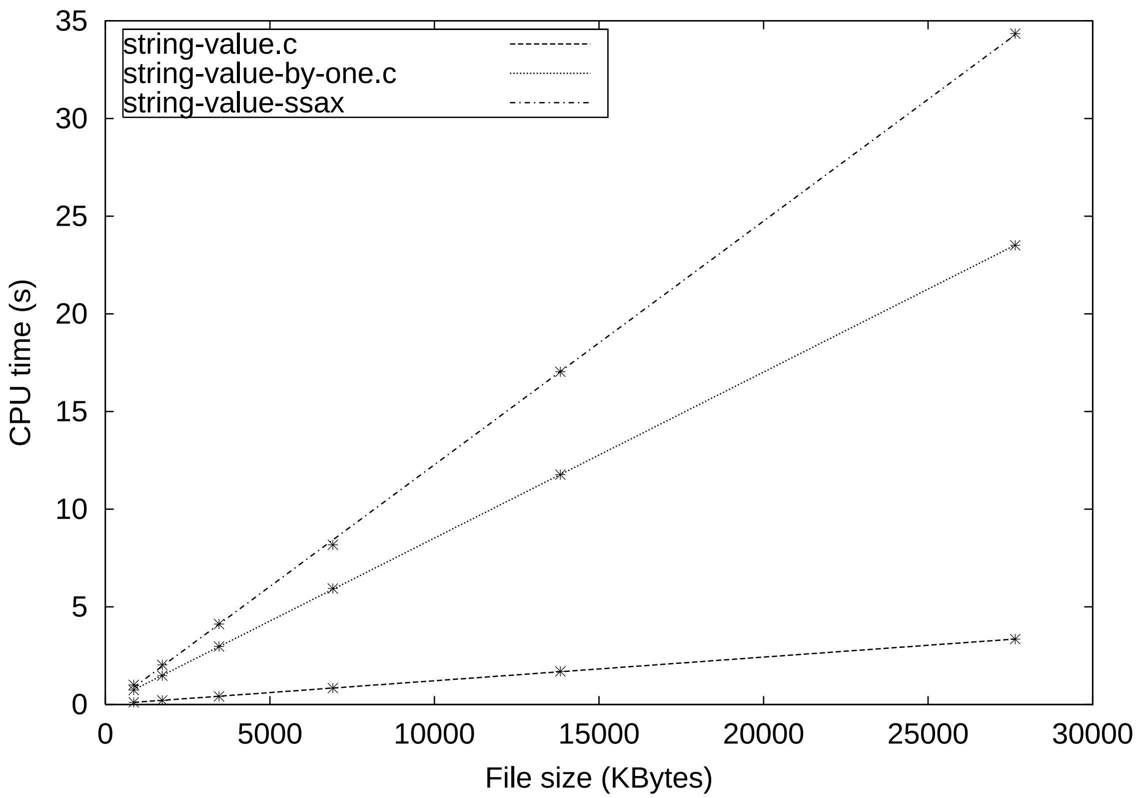

```
<node><node><node><leaf>0</leaf><leaf>1</leaf>
            </node>
            <node><leaf>2</leaf><leaf>3</leaf>
      </node></node>
      ...

str_value = concat . reverse . (str_value' [])
  where
    str_value' = foldts fdown fup fhere
    fdown seed = seed
    fup parent_seed seed = seed
    fhere seed str = str:seed
```

Indeed, untagging is precisely determining the string-value of an XML document tree. *This function* operated on trees represented as linked data structures in memory (like *this*). The function `remove-markup` deals with a tree that is an XML document itself (available through the input port).

<Put this and the previous slide side-by-side>. As you can see, `str_value` and `remove-markup` functions are *identical* modulo parentheses and a few extra arguments. The seed is the list of character data in reverse order. When we encounter character data, we add them to our seed <show on both slides>. The downward action is the identity: it propagates the seed at entrance to the parent to the children. The upward action returns the seed accumulated during child branch traversals <show on both slides>. At the very end, we reverse concatenate the resulting seed. This example makes it very clear that the XML parser engine *is indeed the realization of the extended tree fold*.

We apply this benchmark code (remove-markup) to XML documents that look exactly like *this*: realizations of binary trees of increasing depth. Parsing of such documents – with really a lot of deeply nested markup – is a good exercise and a test of the parsing engine. It is also a good model of a typical Web service reply processing. Here are results.

# Performance results




| File size | 13.5MB |
|---|---|
| string-value.c | 1.6964 s |
| string-value-by-one.c | 11.7658 s |
| string-value-ssax.scm | 17.0322 s |

Pentium III Xeon 500 MHz/128 MB; FreeBSD 4.0-RELEASE; Bigloo Scheme 2.4b

The X axis measures the sizes of the input documents, in KBytes. The documents are the realizations of full binary trees of depths 15 through 20. The Y axis is the user CPU time in seconds for the benchmark code, using Expat or SSAX parsing engines. These two lines correspond to Expat – as expected, Expat parsing time grows linearly with the size of the document. We saw that the naive pure functional traversal of trees with tree fold has O(n logn) complexity. The extended tree fold and the SSAX parser avoid this trap. As the plot shows, SSAX also has a linear time complexity. The paper shows that the garbage complexity of SSAX is linear as well. This is the *experimental* result, obtained by measuring the performance of the full-scale XML parser.

**{**We ran all benchmarks on a Pentium III Xeon 500 MHz computer with 128 MB of main memory and FreeBSD 4.0-RELEASE operating system. The benchmark Scheme code was compiled by a Bigloo 2.4b compiler.**}** The table shows the results for one particular tree, of depth 19 <show as a line on a graph>.

**{**To meaningfully compare SSAX and Expat, we need to discuss the difference in input modes of the two parsers. An application that uses Expat is responsible for reading an XML stream by blocks and passing the blocks to Expat, specifically noting the last block. Expat requires the calling application be able to determine the end of the XML document stream *before* parsing the stream.

If an application can do that, it can read the stream by large blocks. An application can potentially load the whole document into memory and pass this single block to Expat. Expat uses shared substrings extensively, and therefore is specifically optimized for such a scenario. If we take a document from a (tcp) pipe, it may be impossible to tell offhand when to stop reading. Furthermore, if we unwittingly try to read a character past the logical end of stream, we may become deadlocked. SSAX reads ahead by no more than one character, and only when the parser is positive the character to read ahead must be available. SSAX can therefore safely read from pipes, can process sequences of XML documents without extra delimiters, and can handle selected parts of a document.**}**

The first benchmark application, `string-value.c`, implements the most favorable to Expat scenario: it reads the whole document into memory, passes it to Expat and asks the parser to remove the markup. The second benchmark application, `string-value-by-one.c`, also uses Expat and also loads the whole document into memory first. The application however passes the content of that buffer to Expat one character at a time. This simulates the work of the SSAX parser. Finally, a SSAX benchmark `string-value-ssax.scm` likewise loads an XML document first, opens the memory buffer as a string port and passes the port to SSAX.

The most notable result of the benchmarks is that a Scheme application is only 45% slower than a comparable well-written C application, `stringvalue-by-one.c`.

SSAX seems quite competitive in performance, especially keeping in mind that the parser and all of its handlers are referentially transparent. The ability to read from pipes and streams whose end is not known ahead of parsing costs performance. We do think however that the feature is worth the price. Shared substrings, present in some Scheme systems (alas not in the compiler used for benchmarking) will mitigate the trade-off.

Conclusions

- XML parsing as tree traversal

- foldts

$\Longrightarrow$

- Better abstraction of XML parsing

- Better API for an event-based XML parser: callbacks ...
  - have less state
  - referentially-transparent
  - easier to write and to reason about

Thus we have shown an example of a principled construction of a SAX XML parser. The parser is based on a view of XML parsing as a depth-first traversing of an input document considered as a spread-out tree. We have considered the problem of efficient functional traversals of abstract trees and of capturing the pattern of recursion in a generic and expressive control structure: `foldts`. Unlike the regular tree fold, `foldts` permits space- and time-optimal accumulating tree traversals.

The `foldts` operator became the core of a stream-oriented, event-based XML parser. Its engine effectively abstracts the details of traversing the XML document tree; the engine makes it unnecessary for user handlers to maintain their own stack of open elements. The comparison with other SAX parsers (the reference XML parser Expat and its functional analogue FXP) shows that SSAX indeed provides a higher-level abstraction for event-based XML parsing. The user-handlers of SSAX are referentially transparent. All together, this makes callbacks easier to write and thus removes whole classes of possible bugs. These benefits are direct outcome of the principled, declarative approach to XML parsing.

We have nothing to be ashamed of

And another conclusion: we do not have to apologize for the languages and design tools we use (namely, declarative languages and Mathematics). I think analysts working with nearly meaningless UML, cowboy programmers who make the lack of a design a virtue, and C and especially Visual C++ programmers have to apologize – for contradictory specifications, excruciating APIs and buffer overflows. And pay for them from their own pocket.

Backup slides

Breadth-first traversal via `foldts`

```
data Tree = Leaf String | Nd [Tree]

breadth_first =
   foldr (x seed -> foldl (flip (:)) seed x) []
   . foldts fdown fup fhere [[]]
  where
   fhere (level_here:levels_below) str
       = (str:level_here):levels_below
   fdown (_:[]) = []:[]
   fdown (_:levels_below) = levels_below
   fup (level_here:levels_below) new_levels_below
       = level_here:new_levels_below

breadth_first
  (Nd [Leaf "0", Nd [Leaf "11", Leaf "12"],
       Leaf "2"])
```

$\Longrightarrow$ ["0","2","11","12"]

Note that

```
foldr (x seed -> foldl (flip (:)) seed x) []
```

is an efficient implementation of

```
concat . map reverse
```

You can't efficiently implement the breadth_first with the first-order fold (and the second-order fold is inefficient). This was the subject of Gibbons and Jones' paper.

Traversing linked trees

```
data Tree = Leaf String | Ref String | Nd [Tree]
type EntityGrove = [(String, Tree)]

foldtsl entities fdown fup fhere seed (Leaf str)
    = fhere seed str
foldtsl entities fdown fup fhere seed (Ref ref)=
  case lookup ref entities of
    Nothing -> error $ "Undefined entity "++ref
    Just tree | tree == singleton ->
        error $ "Circular ref to entity "++ref
              | otherwise ->
        foldtsl ((ref,singleton):entities)
                fdown fup fhere seed tree
   where
     singleton = Ref "---under processing---"
foldtsl entities fdown fup fhere seed (Nd kids)=
   fup seed $
   foldl (foldtsl entities fdown fup fhere)
         (fdown seed) kids
```

The Tree datatype introduced in Section 2.1 had another kind of leaf: a Ref String. The String represents a "reference" into an EntityGrove. The "Ref String" leaf is to model general entity references of XML. According to the XML Recommendation, an XML document is composed of units called entities. An entity may refer to other entities to cause their inclusion in the document. **{**The replacement text for a (general) entity reference is defined either in DTD (for internal parsed entities) or in separate files. A validating parser replaces entity references with their replacement text. A non-validating XML processor is permitted to keep external entity references unexpanded, delegating the task of fetching and parsing of entities to an application. A data model of an XML document – XML Information Set (Infoset) – defines a special information item for an unexpanded entity reference. We model such information items with the "Ref String" nodes.**}**

The most familiar entity references are ">", "<" etc., which represent characters that may not appear literally in the content of any node. General entities may be far more complex than one-character strings. The replacement text of a general entity may be any well-formed XML fragment. In particular, the replacement text of an entity may contain a reference to another entity. The XML Recommendation specifically forbids circular references. If a circular entity reference does occur, it must be detected and reported, as a violation of a "No Recursion" well-formedness constraint.

The previous examples of computing a string value and a tree digest extend painlessly to linked trees – the trees that may contain "references" to other linked trees. This ease of extension shows the benefits of separating the traversal pattern from node operations. We only need to generalize our traversal combinator, foldts, as follows: We have added an EntityGrove argument for the associative list of declared entities, and a case for a "Ref str" node. The foldtsl code correctly detects the two possible errors: a reference to an undefined entity and a cyclic entity reference.

Enforcement of the "No Recursion" well-formedness constraint is noteworthy. Detection of circular references obviously requires some kind of a list of "active" entities, the ones being currently expanded. Whenever an entity reference is encountered, we check if the entity is already mentioned in the list of active entities. If it does, we report a recursive entity reference error. Otherwise, we add the name of the entity to the list of active entities and begin traversing the "replacement tree" for the entity. *After the traversal is finished*, we need to remove the name of the entity from the active entities list. This essentially imperative recipe is literally implemented in a well-known, Reference XML parser Expat. Expat maintains a dedicated stack of active entities. It is obvious but crucially important that pushing an entity onto the stack of active entities be matched by popping the entity of the stack when its expansion completes. These two stack operations are textually separated by entity expansion code, which may have branches and even non-local exits. It requires a special effort from a developer to make sure the stack popping operation occurs in any path through the entity reference processing code. The pure functional code foldtsl makes it impossible to "overlook" the entities stack popping operation. For one thing, there is no dedicated stack of active entities. We use the list of known entities to mark active entities as well: we associate the name of an active entity with a singleton expansion tree. The singleton tree is to be chosen in such a way so it cannot occur as an expansion tree of any legal entity. An active entity is unmarked *automatically*, when the traversal of its expansion tree finishes. There is absolutely no need for any clean-up operations.

An XML parser requires maintenance of a notable amount of state – a list of active entities, a stack of XML Namespaces, etc. Implementing the parser in a declarative way similar to that of foldtsl makes maintenance of such state far easier and eliminates the whole class of possible bugs.