

## Introduction

HXML is a non-validating XML parser written in Haskell. It is designed for space-efficiency, taking advantage of lazy evaluation to reduce memory requirements. HXML may be used as a drop-in replacement for the HaXml parser in existing programs.

*Availability:* The HXML home page is <http://www.flightlab.com/~joe/hxml>. The current version may be downloaded from <http://www.flightlab.com/~joe/hxml/hxml-0.2.tar.gz>. More recent snapshots may be available in <http://www.flightlab.com/~joe/downloads/>. A hardcopy version of this Web site is also available. (Many thanks to Patryk Zadarnowski for the amazing lambdaTeX package).

The current version is a 0.2 snapshot, and is beta quality. In particular, the interface has not yet stabilized and is subject to change. HXML has been tested with GHC 5.02, NHC 1.12, NHC 1.10 (with 'arrow' patch), and various recent versions of Hugs 98.

Please contact Joe English ([jenglish@flightlab.com](mailto:jenglish@flightlab.com)) with any questions, comments, or bug reports.

## Contents

<b>1</b>	<b>Status</b>	<b>2</b>
1.1	Changes since 0.2 release . . . . .	2
1.2	Changes in version 0.2 . . . . .	2
1.3	Known bugs . . . . .	2
1.4	To do . . . . .	3
<b>2</b>	<b>Data structures</b>	<b>3</b>
2.1	XML Information Set . . . . .	3
2.2	Unsupported InfoSet properties . . . . .	4
<b>3</b>	<b>Trees</b>	<b>4</b>
<b>4</b>	<b>Navigable trees</b>	<b>5</b>
4.1	Single-step navigation . . . . .	6
4.2	Multi-step navigation . . . . .	6
<b>5</b>	<b>Simplified representation</b>	<b>6</b>
<b>6</b>	<b>High-level interface</b>	<b>6</b>
6.1	Node tests . . . . .	7
6.2	Navigation . . . . .	7
6.3	Constructors . . . . .	8
6.4	Editing combinators . . . . .	9
<b>7</b>	<b>Arrows</b>	<b>9</b>

1	Status	2
7.1	Combining arrows: product combinators . . . . .	10
7.2	Adding guards: class ArrowZero . . . . .	11
7.3	Adding conditionals: class ArrowChoice . . . . .	11
8	HaXml adapter	12
9	Low-level interface	13
10	Public Modules	14
11	Internal modules	14

## 1 Status

### 1.1 Changes since 0.2 release

- Added simplified XML representation `ELEMENT STRING ATTLIST | TEXT STRING` that strips out the inessential Information Set items; see module `Etree`

### 1.2 Changes in version 0.2

- Added Arrow-based combinator library
- Added support for CDATA sections
- New function `parseDocument` recognizes (and ignores) the document prolog (XML and DOCTYPE declarations)
- Several data structures and public functions have been renamed
- Space fault in comment parsing fixed

### 1.3 Known bugs

- The XML declaration is ignored
- Unicode support is only as good as that provided by the Haskell system (i.e., not very, except for HBC)
- Does not support XML Namespaces.
- Does not do any well-formedness or validity checks

## 1.4 To do

- Rewrite `doc/mkSite.hs` as a Literate Haskell script
- Add Filter combinators that operate on the `NTree` representation

## 2 Data structures

In HXML, XML documents are represented as a Tree of XMLNodes:

```

type NAME = STRING
type ATTLIST = [(NAME,STRING)]

data XMLNODE =
    RTNODE
  | ELNODE NAME ATTLIST
  | TXNODE STRING
  | PINODE NAME STRING
  | CXNODE STRING
  | ENNODE NAME
data TREE a = TREE a [TREE a]
type XML = TREE XMLNODE

parseXML :: STRING → XML
showXML  :: XML → STRING

```

Name is a type synonym for `String`; it is used for element and attribute names. The `AttList` type denotes an attribute value list, represented in HXML as a list of (*name*, *value*) pairs.

### 2.1 XML Information Set

`XMLNode` is an algebraic data type roughly corresponding to XML Information Set items, (<http://www.w3.org/TR/xml-infoset>) as follows:

#### **RTNode**

The document information item. This node type is used as a container node to hold the result of `parseDocument`. In addition to the root element, it may contain processing instructions and other miscellaneous junk.

#### **ELNode GI AttList**

An element information item. *GI* is the generic identifier or element type name. *AttList* is the list of attribute information items belonging to this element, represented as a list of name-value pairs [(`String`, `String`)].

HXML

**TXNode String**

A consecutive sequence of character information items. Note that a TXNode does not necessarily represent a *maximal* contiguous sequence of characters – the parser may split text up into multiple TXNodes.

**PINode Name String**

A processing instruction information item. The first field is the processing instruction *target* property, the second is its *content* property. The *notation* processing instruction property is not included; I believe this is an error in the InfoSet spec.

**CXNode String**

A comment information item.

**ENNode Name**

An unparsed entity information item. Generated by a named entity reference (*&foo;*).

## 2.2 Unsupported InfoSet properties

The following properties defined in the InfoSet are not directly supported by HXML:

- base URI
- character encoding scheme
- standalone
- namespace name
- namespace attributes
- in-scope namespaces
- attribute type
- references
- ... probably others

Other infoset properties such as *parent* and *children* are derived from the *Tree* context. The *prefix* and *local name* properties of named nodes (elements, attributes, etc.) may be extracted from the name with the *splitName* function. Most of the additional unparsed entity information item properties are available only with great effort.

## 3 Trees

Module *Tree* defines several general-purpose functions for operating on trees.

```
treeRoot :: TREE a → a
treeChildren :: TREE a → [TREE a]
preorderTree :: TREE a → [a]
```

`treeRoot` and `treeChildren` are the projection functions, returning the root value and list of subtrees respectively. `preorderTree` returns a list of all values stored in the tree.

The usual polytypic functions are available, which do the obvious things:

```
mapTree :: (a → b) → TREE a → TREE b
cataTree :: ((a, [b]) → b) → TREE a → b
anaTree :: (b → (a, [b])) → b → TREE a
instance FUNCTOR TREE where fmap = mapTree
```

`foldTree` is the Tree analogue of the list function `foldr`. `accumTree` is a shape-preserving downwards accumulation; it takes a seed value and a combining function  $f :: a \rightarrow b \rightarrow (c, a)$ , which it evaluates at each level of the tree to produce a new node value and seed. `scanTree` is a variant of `accumTree`; it's analogous to the list function `scanl`.

```
foldTree :: (a → b → c) → (c → b → b) → b → TREE a → c
scanTree :: (a → b → a) → a → TREE b → TREE a
accumTree :: (a → b → (c, a)) → a → TREE b → TREE c

foldTree tree cons nil (TREE a c)
    = tree a (foldr cons nil (map (foldTree tree cons nil) c))
scanTree op a (TREE b children)
    = let a' = a `op` b in TREE a' (map (scanTree op a') children)
accumTree op a (TREE b children)
    = let (c, a') = a `op` b in TREE c (map (accumTree op a') children)
```

## 4 Navigable trees

With `type XML = TREE XMLNODE`, it is possible to access the children and descendants of a node, but not the ancestors or siblings. Module `NTree` provides a “navigable tree” data structure that can traverse up, left, and right as well as down the tree. It comes at the price of space-efficiency though: using an `NTree` instead of a `Tree` means that the entire document must be kept in memory and cannot be garbage-collected.

```
type NTREE a = ... {- opaque -}
nTree :: TREE a → NTREE a
subtreeNT :: NTREE a → TREE a
dataNT :: NTREE a → a
```

The `nTree` function converts a `TREE` into an `NTREE`. `subtreeNT` goes the other way, extracting the `TREE` value at the current node. `dataNT` returns the label of the current node; it is equivalent to `treeRoot . subtreeNT`.

## 4.1 Single-step navigation

*upNT*, *downNT*, *leftNT*, *rightNT* :: NTREE *a* → MAYBE (NTREE *a*)

*upNT* returns the parent of the input node; *downNT* returns the first child of the input node. *leftNT* (resp. *rightNT*) return the previous (resp. next) sibling node.

The return value for all four functions is MAYBE (NTREE *a*); they return NOTHING if the input node is, respectively, the root, a leaf, or the first or last child of its parent.

## 4.2 Multi-step navigation

All of the principal XPath axes (excluding *attribute::\** and *namespace::\**) are provided:

*ancestorAxis*, *ancestorOrSelfAxis*, *childAxis*,  
*descendantAxis*, *descendantOrSelfAxis*,  
*followingAxis*, *followingSiblingAxis*, *parentAxis*,  
*precedingAxis*, *precedingSiblingAxis*, *selfAxis*  
 :: NTREE *a* → [NTREE *a*]

## 5 Simplified representation

For applications which do not need all the details in the full XML Information Set, module *ETree* provides a simplified representation:

```
data ETree
  = ELEMENT NAME ATTLIST [ETree]
  | TEXT STRING

xmlToETree :: XML → ETree
etreeToXML :: ETree → XML
```

## 6 High-level interface

Module *XMLCombinators* provides a high-level XML programming interface. Like *HaXML*, it is based on gluing together *Filters*, where a *Filter* is a function from nodes to lists of nodes, or more generally from any type *a* to lists of any other type *b*

HXML

```

newtype FILTER a b = FILTER (a → [b])

runFilter :: FILTER a b → a → [b]
makeFilter :: (a → [b]) → FILTER a b
apFilter :: ([a] → [b]) → FILTER c a → FILTER c b
aEach :: FILTER [a] a

```

The `runFilter` function applies a filter to an input to produce a list of outputs; `makeFilter` turns a function  $f :: a \rightarrow [b]$  into a Filter  $makeFilter\ f :: FILTER\ a\ b$ . (These two functions are inverses; in fact both are the identity.)

`apFilter` applies a list function  $f :: [a] \rightarrow [b]$  to a filter to produce a new filter; `apFilter func filt` is shorthand for `makeFilter (func . runFilter filt)`.

## 6.1 Node tests

The node test filters are *guards*; they filter out nodes which fail to satisfy some predicate.

```

qElem :: STRING → FILTER XML XML
qElems :: [STRING] → FILTER XML XML
qText, qEntity :: FILTER XML XML
qMatch :: FILTER x y → FILTER x x

```

`qElem` selects element nodes with a specified element type name. `qElems` takes a list of element type names, and selects element nodes with any of those names. `qText` and `qEntity` select text nodes and entity reference nodes, respectively.

`qMatch` turns an arbitrary filter into a guard: `qMatch f` succeeds if `f` returns a nonempty list when applied to the input.

## 6.2 Navigation

The following filters navigate down Tree structures:

```

qSelf, qChildren, qDescendants, qSubtree :: FILTER (TREE a) (TREE a)

```

`qSelf` is the identity arrow; it corresponds to the XPath `self` axis. `qChildren` returns the immediate children of the input node, and `qDescendants` returns all proper descendants. `qSubtree` returns all of the nodes rooted at the input node; it is equivalent to the XPath `descendants-or-self` axis.

`|>|` is the “distinguished choice” operator:  $f\ |>|\ g$  returns the result of evaluating `f` if the result is nonempty, otherwise the result of evaluating `g`.

`qOuter` recursively descends the tree, applying a filter at each step, and returns the outermost nonempty result. `qInner` is similar, but proceeds from the bottom up:

```

qInner, qOuter :: FILTER (TREE a) b → FILTER (TREE a) b
qOuter p = p |>| (qChildren >>> qOuter p)
qInner p = (qChildren >>> qInner p) |>| p

```

`qFirst` applies a filter and returns only the first result:

```

qFirst :: FILTER a b → FILTER a b
qFirst f = apFilter take1 where
    take1 (x:_) = [x]
    take1 [] = []

```

`qAttributes` is an XML-specific filter that returns the attribute list of the input node. `qAttribute` returns the value of a specific attribute. `qNodeName` returns the node name of the current node; this is the element type name for elements, PI target for processing instructions, or entity name for entity reference nodes. For other node types, returns nothing.

```

qAttributes :: FILTER XML (NAME, STRING)
qAttribute  :: NAME → FILTER XML STRING
qNodeName  :: FILTER XML NAME

```

## 6.3 Constructors

```

mkElem :: STRING → FILTER a XML → FILTER a XML
mkElement :: STRING → FILTER a (NAME,STRING) → FILTER a XML → FILTER a XML
mkText  :: FILTER STRING XML
mkLiteral :: STRING → FILTER a XML
mkAtt   :: NAME → FILTER a STRING → FILTER a (NAME,STRING)

```

`mkElement name atts content` is a filter which creates an element node with the specified *name*; it passes its input to the *atts* filter to construct the attribute list, and to the *content* filter to construct the list of children. `mkElem` is a shorthand version of `mkElement` for the common case where there are no attributes; it is defined as `mkElem gi body = mkElement gi aZero body`.

`mkAtt` constructs an attribute value (represented in HXML as a *(name, value)* pair).

`mkLiteral` and `mkText` both construct text nodes. `mkText` is a filter which creates a text node containing its (string) input. `mkLiteral txt` is equivalent to `aConst txt >>> mkText`.

## 6.4 Editing combinators

```
apChildren, aFoldTree, aScanTree
  :: FILTER (TREE a) (TREE a) → FILTER (TREE a) (TREE a)
```

*apChildren* applies a specified filter to the children of the input tree, and replaces them with the result. *aFoldTree* and *aScanTree* recursively apply a filter to each node in the tree, *aFoldTree* from the bottom up and *aScanTree* from the top down:

```
aFoldTree f = apChildren (aFoldTree f) >>> f
aScanTree f = f >>> apChildren (aScanTree f)
```

## 7 Arrows

HXML Filters are an instance of the *Arrow* class.

Arrows are much easier to use than to explain. See <http://www.soi.city.ac.uk/~ross/arrows/> for a longer discussion of arrows (but note that the HXML implementation uses a different notation for many of the combinators).

```
class ARROW a where
  (>>>) :: a b c → a c d → a b c
  arr :: (b → c) → a b c
```

Basically, the *Arrow* type class represents a computation that takes inputs of some type *b* and produces outputs of some other type *c*. Several combinators are available to glue together arrows. The *>>>* operator composes two arrows in sequence: if *f* :: *FILTER a b* and *g* :: *FILTER b c*, then *f >>> g* :: *FILTER a c* passes the output of *f* to the input of *g*.

The *arr* combinator turns an ordinary function *f* :: *b* → *c* into an arrow *arr f* :: *a b c*. *aConst* constructs a constant arrow, which always outputs a specified value: *aConst x = arr (const x)*.

The simplest kind of Arrows are ordinary functions: here *>>>* is just function composition (with the arguments in the opposite order from usual), and *arr* is the identity:

```
instance ARROW (→) where
  f >>> g = λx → g (f x)
  arr f = f
```

Any Monad *m* can be made into a monadic arrow *MA m* (also called a Kleisli arrow).

```
newtype MA m a b = MA (a → m b)
```

```

instance MONAD m ⇒ ARROW (MA m) where
  arr f = MA (return . f)
  MA f >>> MA g = MA (λb → f b >>= g)

```

HXML Filters are another kind of arrow: here composing two arrows  $f$  and  $g$  applies  $f$  to an input to produce a list of intermediate results, applies  $g$  to each intermediate result, and concatenates the output of  $g$  to produce the final result. Filters are equivalent to monadic arrows over the List monad, `MA []`, though the implementation is slightly different.

```

instance ARROW FILTER where
  arr f = FILTER (λx → [f x])
  (FILTER f) >>> (FILTER g)
    = FILTER (λx → [z | y ← f x, z ← g y])
    = FILTER (concatMap f g)

```

The `+++` operator applies two filters to the same input and concatenates their result:

```

class ARROWPLUS a where
  (+++) :: a b c → a b c → a b c
instance ARROWPLUS FILTER where
  (FILTER f) +++ (FILTER g) = FILTER (λx → f x ++ g x)

```

## 7.1 Combining arrows: product combinators

```

class ARROW a where
  (&&&) :: a b c → a b d → a b (c,d)
  (>&<) :: a b c → a d e → a (b,d) (c,e)
  apfst :: a b c → a (b,x) (c,x)
  apsnd :: a b c → a (x,b) (x,c)

```

The `&&&` operator is a Cartesian product combinator:  $f \&\&\& g$  passes its input to both  $f$  and  $g$ , and combines the results.  $>\&<$  combines two arrows in a different way:  $f >\&< g$  takes a pair as input, passes the first element to  $f$ , the second element to  $g$  and combines the results. Finally, `apfst` and `apsnd` apply a process to the first (resp. second) member of their input and pass the other on unchanged.

This is probably best understood by looking at the instance definitions for function and filter arrows:

```

instance ARROW (→) where
  f &&& g = λb → (f b, g b)
  f >&< g = λ(b,d) → (f b, g d)
  apfst f (b,c) = (f b,c)

```

HXML

```

        apsnd g (b,c) = (b,g c)
instance ARROW FILTER where
    (FILTER f) &&& (FILTER g)
        = FILTER $ \b → [(c,d) | c ← f b, d ← g b]
    (FILTER f) >&< (FILTER g)
        = FILTER $ \(b,d) → [(c,e) | c ← f b, e ← g d]

```

## 7.2 Adding guards: class ArrowZero

An ArrowZero is a kind of arrow that has a natural “zero”, “failure”, or “nothing” operation, `aZero`. For Filter, this is (the arrow that always returns) the empty list. ArrowZero also supports *guards*, which pass inputs which satisfy some predicate through unchanged; those which fail the test are mapped to the zero element. The `aMaybe` arrow is a convenience routine that “unwraps” a Maybe input, mapping `JUST x` to `x` and `NOTHING` to the zero element.

```

class (ARROW a) ⇒ ARROWZERO a where
    aZero :: a b c
    aGuard :: (b → BOOL) → a b b
    aMaybe :: a (MAYBE c) c

```

## 7.3 Adding conditionals: class ArrowChoice

The ArrowChoice class enhances arrows with conditional expressions, using the standard Haskell disjoint union type `Either`:

```

class (ARROW a) ⇒ ARROWCHOICE a where
    (|||) :: a b c → a d c → a (EITHER b d) c
    (>|<) :: a b c → a d e → a (EITHER b d) (EITHER c e)
    apl :: a b c → a (EITHER b d) (EITHER c d)
    apr :: a b c → a (EITHER d b) (EITHER d c)

```

— ...

```

instance ARROWCHOICE (→) where
    f (|||) g = \x → case x of LEFT b → f b; RIGHT d → g d
    f (>|<) g = (LEFT . f) ||| (RIGHT . g)
    apl f = f >|< id
    apr g = id >|< g

```

The sum combinators `|||`, `>|<`, `apl`, and `apr` are dual to the product combinators `&&&`, `>&<`, `apfst`, and `apsnd`.

There is also an if-then-else construct analogous to C's ternary `... ? ... : ...` operator:

```
data CHOICE a = a :> a
class (ARROW a) => ARROWCHOICE a where
  (?>) :: (b -> BOOL) -> CHOICE (a b c) -> a b c
  (>?) :: a b BOOL -> CHOICE (a b c) -> a b c
```

```
instance ARROWCHOICE (->) where
  p ?> f :> g = \x -> if p x then f x else g x
```

If  $p$  is a predicate  $p :: b \rightarrow \text{BOOL}$  and  $f$  and  $g$  are arrows  $f, g :: \text{ARROW } b \ c$ , then  $p ?> f :> g$  passes its input to one of  $f$  or  $g$  depending on the result of  $p$ . For the  $>?>$  combinator, the condition is also an arrow  $p :: \text{ARROW } b \ \text{BOOL}$ .

## 8 HaXml adapter

The `HaXmlAdapter` module contains modified versions of `HaXml`'s `processXmlWith` driver:

```
processXmlWith' :: (CONTENT -> [CONTENT]) -> IO ()
processXmlWith'' :: (CONTENT -> [CONTENT]) -> IO ()
```

The first uses the `HXML` parser instead of the `HaXml` one; the second in addition uses `HXML`'s serializer. These versions should have better space performance than `processXmlWith` (although there are still a few problems).

*NOTE:* To avoid a space fault in `HaXml` version 1.02 and earlier, you may wish to replace the definition of `cat` in `XmlCombinators.hs` with the following implementation:

```
cat :: [a -> [b]] -> (a -> [b])
cat [] = const []
cat l = foldr1 (lift (++)) l
```

With this patch and `processXmlWith''`, the space requirements of many existing `HaXml` programs may be significantly reduced.

Adapters for converting to and from `HaXml`'s internal representation (`Content`) are also available, along with a few utilities:

```
toContent :: TREE XMLNODE -> CONTENT
fromContent :: CONTENT -> TREE XMLNODE

buildContent :: [XMLEVENT] -> CONTENT
```

HXML

```

serializeContent :: CONTENT → [XMLEVENT]

parseContent :: STRING → CONTENT
showContent :: CONTENT → STRING

showContent = showXML . serializeContent
parseContent = buildContent . parseInstance

```

## 9 Low-level interface

Module *XMLParser* defines the low-level interface to the parser.

```

parseDocument :: STRING → [XMLEVENT]
parseInstance :: STRING → [XMLEVENT]

```

*parseInstance* parses an XML instance into a list of events. *parseDocument* does the same, but parses the document prolog (optional XML and DOCTYPE declarations) as well. *XMLEvent* is defined in *XMLParse.hs* as:

```

type NAME = STRING
data XMLEVENT =
  STARTEVENT NAME [(NAME,STRING)] — start-tag
  | EMPTYEVENT NAME [(NAME,STRING)] — empty element
  | ENDEVENT NAME — end-tag
  | TEXTEVENT STRING — character data
  | PIEVENT NAME STRING — processing instruction
  | GEREFEVENT NAME — general entity reference
  | COMMENTEVENT STRING — comment
  | ERROREVENT STRING — error report

```

This provides a “SAX-like” interface, or rather the FP equivalent of SAX. Instead of invoking callback methods on a handler object, the parser returns a (lazy) list of events.

*parseInstance* is most usefully composed with

```

buildTree :: [XMLEVENT] → TREE XMLNODE

```

defined in module *TreeBuild*. This module also defines the converse operation,

```

serializeTree :: TREE XMLNODE → [XMLEVENT]

```

These are not inverses – *serialize* . *build* is not the identity – but they should satisfy

```

build . serialize . build = build
serialize . build . serialize = serialize

```

## 10 Public Modules

### **module HXML**

Package module which simply includes and reexports the HXML public modules listed below.

### **module XML**

Defines data types for XML document instances.

### **module DTD**

Defines data types for XML (and SGML) DTDs

### **module XMLParse**

The parser.

### **module Tree**

Definition of Rose trees, **data** `TREE a = TREE a [TREE a]`, plus a few miscellaneous utilities.

### **module TreeBuild**

Converters between the stream and tree representations.

### **module PrintXML**

The serializer: converts XML (tree view or stream view) back into a sequence of characters. `showXML` serializes an XML tree as a string; `showEvents` serializes a sequence of `XMLEvents`.

```
showXML :: TREE XMLNODE → STRING
```

```
showEvents :: [XMLEVENT] → STRING
```

## 11 Internal modules

### **module LLParsing.hs**

CPS-based parser combinator library used by XMLParse. Uses function application for sequencing a la Swierstra and Duponcheel; the user interface is similar to Swierstra & Duponcheel's `UU_Parsing`, but the implementation is much simpler (and less powerful).

This is the only part of the code that requires an extension to Haskell 98 (rank-2 polymorphism), and that can be avoided simply by replacing `newtype Parser sym res = P ...` with `newtype P p = P p` and omitting all other type declarations involving `Parser`. Haskell's type inference algorithm computes workable (though incomprehensible) types for all the combinators.

### **module XMLScanner**

The lexical analyzer. Parses a sequence of characters into a sequence of Delimiters. In addition to the `Delimiter` data type (q.v.), exports two functions: `pcdataMode :: STRING → [DELIMITER]` and `markupMode :: STRING → [DELIMITER]`. `pcdataMode` is used to parse the document instance, and `markupMode` is used to parse DTDs.

### **module AssocList**

A quick and dirty finite map implementation; used by `DTD.hs`

HXML

**module Misc**

Miscellaneous handy utilities which didn't fit anywhere else.

**Index**

- +++ , 10
- &&&, 10, 12
- |||, 12
- |>|, 7
- >?>, 12
- >&<, 10, 12
- >|<, 12
- >>>, 9
  
- accumTree, 5
- aConst, 9
- aFoldTree, 9
- aMaybe, 11
- apChildren, 9
- apFilter, 7
- apfst, 10, 12
- apl, 12
- apr, 12
- apsnd, 10, 12
- arr, 9
- Arrow, 9
- ArrowChoice, 11
- ArrowZero, 11
- aScanTree, 9
- AttList, 3
- attribute information items, 3
- aZero, 11
  
- cat, 12
- character information items, 4
- children, 4
- comment information item, 4
- Content, 12
- content, 4
  
- dataNT, 5
- Delimiter, 14
- descendants-or-self, 7
- document information item, 3
- downNT, 6
  
- Either, 11
  
- element information item, 3
- ETree, 6
- Etree, 2
  
- Filter, 6, 10, 11
- foldTree, 5
  
- HaXmlAdapter, 12
  
- Kleisli, 9
  
- leftNT, 6
- local name, 4
  
- makeFilter, 7
- markupMode, 14
- Maybe, 11
- mkAtt, 8
- mkElem, 8
- mkElement, 8
- mkLiteral, 8
- mkText, 8
  
- Name, 3
- node name, 8
- notation, 4
- NTree, 3, 5
- ntree, 5
  
- parent, 4
- parseDocument, 2, 3, 13
- parseInstance, 13
- Parser, 14
- pcdataMode, 14
- prefix, 4
- preorderTree, 5
- processing instruction information item, 4
- processXmlWith, 12
- processXmlWith", 12
  
- qAttribute, 8
- qAttributes, 8

- qChildren, 7
- qDescendants, 7
- qElem, 7
- qElems, 7
- qEntity, 7
- qFirst, 8
- qInner, 8
- qMatch, 7
- qNodeName, 8
- qOuter, 7
- qSelf, 7
- qSubtree, 7
- qText, 7
  
- rightNT, 6
- runFilter, 7
  
- scanl, 5
- scanTree, 5
- self, 7
- showEvents, 14
- showXML, 14
- splitName, 4
- subtreeNT, 5
  
- target, 4
- Tree, 3–5, 7
- TreeBuild, 13
- treeChildren, 5
- treeRoot, 5
  
- unparsed entity information item, 4
- upNT, 6
  
- XMLCombinators, 6
- XmlCombinators.hs, 12
- XMLEvent, 13, 14
- XMLNode, 3