Design and Implementation of a validating XML parser in Haskell

Master's thesis

University of Applied Sciences Wedel

Computer Science Department

Martin Schmidt

Design and Implementation of a validating XML parser in Haskell: Master's thesis;
University of Applied Sciences Wedel
Computer Science Department
by Martin Schmidt

Published 2002-09-02

Abstract

This thesis introduces the core component of the Haskell XML Toolbox: a validating XML parser that supports almost fully the Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation [WWW01]. The thesis presents how a validating XML parser and XML processing applications can be implemented by using filter functions as a uniform design.

The Haskell XML Toolbox is a collection of tools for processing XML with Haskell. It is itself purely written in Haskell. The Toolbox is a project of the University of Applied Sciences Wedel, initialized by Prof. Dr. Uwe Schmidt.

The Haskell XML Toolbox bases on the ideas of HaXml [WWW21] and HXML [WWW25], but introduces a more general approach for processing XML with Haskell. It uses a generic data model for representing XML documents, including the DTD subset and the document subset. This data model makes is possible to use filter functions as a uniform design of XML processing applications. Libraries with filters and combinators are provided for processing this data model.

The following components are included:

- `hdom` - Core data types and functions for processing XML with Haskell
- `hparser` - XML parser
- `hvalidator` - Modules for validating XML documents
- `hxslt` - Modules for XSL transformations

Prof. Dr. Uwe Schmidt wrote the basic parser and core functions. His master student Christine Nickel wrote the package `hxslt`, his master student Martin Schmidt wrote the package `hvalidator` and some parts of the parser.

Related work

Malcolm Wallace and Colin Runciman wrote HaXml [WWW21], a collection of utilities for using Haskell and XML together. The Haskell XML Toolbox is based on their idea of using filter combinators for processing XML with Haskell.

Joe English wrote HXML [WWW25], a non-validating XML parser in Haskell. His idea of validating XML by using derivatives of regular expressions [WWW26] was implemented in the validation functions of this software.

Dedication

"Learn at least one new [programming] language every year.
Different languages solve the same problems in different ways.
By learning several different approaches, you can help broaden
your thinking and avoid getting stuck in a rut."
--- The Pragmatic Programmer

Table of Contents

List of Tables

List of Figures

List of Examples

Preface

The primary aim of this project is to gain some valuable experiences with the beautiful functional programming Haskell. Parsing and processing lists are strengths of functional programming languages. Both features are needed to handle XML. Combined with higher-order combinatorial functions, that a functional programming language like Haskell allows, processing XML can be very efficient and elegant.

Many of the techniques described in this master thesis are far more elegant, compact and powerful than the ones found in familiar techniques like DOM [WWW05] or JDOM [WWW06].

Because there did not exist any validating XML parser written in Haskell, we thought it might be a nice project implementing one and doing some further XML processing with Haskell. The Haskell XML Toolbox uses a general tree data model for representing XML documents. This generic data model makes it possible to implement an XML parser or XML processing applications with a uniform design by using filter functions and combinators for processing XML.

We have chosen Haskell because it is a popular modern functional programming language, and we were interested in learning more about the functional programming paradigm. There exist lots of free implementations, online tutorials and some good Haskell books. The Haskell homepage Haskell.org [WWW11] gives many further details.

Learning functional programming with Haskell by self-studies was very challenging. It required a change in perspective of programming. But once the paradigms are understood, writing Haskell programs is very straightforward and makes a lot of fun.

The first chapter gives an introduction to XML and Haskell. Readers who are familiar with these techniques can skip it. The next three chapters describe the packages `hdom`, `hparser` and `hvalidator` of the Haskell XML Toolbox. The last chapter compares the design used in the Haskell XML Toolbox with HaXml and HXML.

Chapter 1. Basics

The following chapter gives a short introduction to XML and Haskell. Readers who are familiar with these techniques can skip this chapter.

1.1. XML

1.1.1. Introduction

The Extensible Markup Language (XML) was standardized by the World Wide Web Consortium (W3C) at the beginning of 1998. The current standard is described by the Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation [WWW01]. XML forms a subset of the Structured Generalized Markup Language (SGML), which is extremely complex. XML itself is a simplified form of SGML. Both languages are meta languages for defining markup languages.

XML documents are structured hierarchical. The documents base on a simple data model: trees with attributes as some extra structure.

A Document Type Definition (DTD) can be used to define formal requirements to the structure of an XML document. It specifies a grammar for a class of documents. These are rules for the elements, attributes and other data and how they are logically related in an XML document. Each document, which corresponds to a DTD, is called an instance of this DTD.

Today XML is often used to define and exchange data structures application-independent. A central role plays the validation of XML documents. Validation of XML documents describes the check if the document corresponds to the constraints of a DTD. The application that reads an XML document and performs validation is called XML processor. Not all XML processors are validating ones. Applications that do not need these checks can use non-validating XML processors for performance reasons.

1.1.2. Processing XML

Every application that processes information from XML documents needs an XML processor, which reads an XML document and provides access to its content and structure to the processing application. Two parse methods exist: event based and tree based parse method. They define essential differences for the communication between the XML processor and applications.

1.1.2.1. Event based parse method

The document is processed sequentially; the most popular parse method is SAX - the Simple API for XML [WWW10]. Each element is an event trigger, which can initiate an action on the processing application. In terms of processing speed and memory consumption this is the most performant technique for accessing an XML document in sequential order. The event based parsed method is

also very efficient for processing only a few specific elements of a document. The main disadvantage is that it is more difficult to generate output with a different order of elements than the input, because some kind of memory is needed.

1.1.2.2. Tree based parse method

The document is transformed from a sequential into a hierarchical structure by the XML processor (e.g. DOM/JDOM). This tree model of the document is stored in memory and can be accessed by the processing application. The main advantage of this parse method is that it supports random access to the document. Random access is needed for example for Extensible Stylesheet Language Transformations (XSLT) [WWW07]. XSLT implements a tree-oriented transformation language for transmuting XML documents.

The tree representation can be used for traversing the document several times or construct output in a different order from the input. The main disadvantage of this method is that the tree model is only accessible when parsing of the document is complete. For large documents this can be slow and memory intensive.

1.1.3. Correctness of documents

1.1.3.1. Well-formed documents

There exist two classes of correctness for XML documents. The first class form the well-formed documents. These documents meet the syntax rules of the XML 1.0 specification [WWW01]. If a document does not meet these syntax rules, it is not an XML document and is rejected by any XML processor.

Some XML syntax rules:

- Every document must have a single unique root element that encloses all other elements.
- All elements must be correctly nested.
- All elements must have corresponding start and end tags.
- Attribute values must be enclosed within single or double quotes.

1.1.3.2. Valid documents

The second class form the valid documents. These documents are well-formed and meet in addition the constraints of a DTD (described in Section 1.1.4) or a Schema [WWW08]. This class of correctness is required if XML documents must be exchanged between applications reliably. Only validating XML processors perform these checks, non-validating XML processors ignore the DTD.

1.1.4. Document Type Definition (DTD)

XML is a meta language for specifying markup languages, the DTD is a tool to create and describe this language. A DTD defines the elements, attributes, entities and notations used in an XML document. Further it defines a context free grammar that states which is the allowed content of an element. The DTD can point to an external DTD subset containing declarations, it may contain the declarations directly in an internal DTD subset, or even both. This section will describe the definitions of elements and their attributes. They form the basic concepts of XML documents. Entity declarations that are mainly used for replacing text and notation declarations that are used for specifying the format of special contents are not explained. For a complete overview see [WWW04].

1.1.4.1. Element type declaration

A DTD declares all allowed elements in an XML document. Elements are defined by <!ELEMENT> declarations. This declaration specifies the name of the element and its content model. Only one declaration for each element is allowed, multiple declarations are forbidden. The order of the declarations is irrelevant.

`<!ELEMENT name content model>`

- `ELEMENT` - Keyword
- `name` - Element name, respectively its tag names
- `content model` - Definition of the allowed types of child elements (text data, other elements) and their order

For defining the content model of elements there exist some operators. These operators are described shortly in the following.

Table 1-1. Connectors

,	Sequence of elements, "then"
|	Alternative of elements (exclusive), "xor"

Table 1-2. Occurrence indicators

(no indicator)	Element must appear exactly one time
?	Element is optional, 0 or 1
*	Element is optional and repeatable, 0 or more
+	Element must appear at least one time, 1 or more

Table 1-3. Grouping

(	Start of content model or group
)	End of content model or group

Table 1-4. #PCDATA in content model

(#PCDATA)	Element has only text data or nothing as content
(#PCDATA | el1 | el2)*	Mixed content. The element can contain text data and elements. #PCDATA must be listed in the declaration at first. It cannot be defined that allowed child elements have to appear in a certain order, or that they at least must appear.

#PCDATA is the abbreviation for "parsed character data" and means that text data is inspected by the XML parser for eventual markup. If for example the text data contains entity references, the entities are expanded.

Table 1-5. Keywords

EMPTY	Element has no content
ANY	Element has any declared elements and text data as content, or is empty.

1.1.4.2. Attribute declarations

Attributes define additional name-value pairs for elements. The attributes are defined by <!ATTLIST> declarations. It defines the attribute type and default values.

```
<!ATTLIST target_element attr_name attr_type default>
```

- `ATTLIST` - Keyword
- `target_element` - Element name
- `attr_name` - Attribute name
- `attr_type` - Type of the attribute value, or list of values
- `default` - Keyword or default value

Table 1-6. Type of the attribute value

CDATA	Text data
ID	Unique identifier, value of attribute must be unique within the XML document
IDREF	Reference to an ID, must match the value of some ID attribute
IDREFS	One or more references to IDs, separated by white space
NMTOKEN	Name token, attribute value must only consist of letters, digits and some extra characters
NMTOKENS	One or more name tokens, separated by white space
ENTITY	Name of an unparsed entity, declared in the DTD
ENTITIES	One or more names of unparsed entities declared in the DTD, separated by white space
(a | b | c)	Enumeration, list of possible attribute values

Table 1-7. Attribute defaults

"value"	Default value of the attribute. If the attribute is not specified, this value is used by the XML processor
#REQUIRED	Attribute must be specified
#IMPLIED	Attribute is optional
#FIXED "value"	Attribute has a fixed value. No other value is allowed.

1.1.5. Example

This simple example demonstrates an XML document with an internal DTD. The DTD defines the elements `a`, `b` and `c`. The element `a` has an additional attribute `att1`. The small document enclosed by the root element `a` is an instance of this DTD.

Example 1-1. Sample XML document

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<!DOCTYPE a [
<!ATTLIST a  att1  CDATA  #IMPLIED>
<!ELEMENT a  (b, c?)>
<!ELEMENT b  EMPTY>
<!ELEMENT c  (#PCDATA)>
]>

<a att1="test">
    <b/>
    <c>hello world</c>
</a>
```

1.2. Haskell

This introduction to Haskell can only cover a few aspects of Haskell. Its aim is to make the next chapters more easily understandable for persons who are not familiar with Haskell, but with programming languages. If you are new to Haskell, its homepage Haskell.org [WWW11] is a good place to start for gaining information and lots of free compilers and libraries. The tutorial "A Gentle Introduction to Haskell" [WWW14] gives a detailed overview about the language. A great learning book, full of examples is "The Craft of Functional Programming" by Simon Thompson [Thompson99]. For advanced people Paul Hudak's "The Haskell School of Expression" [Hudak00] gives a more complete overview about monads and higher-order functions by samples from multimedia. The language itself is defined in the Haskell 98 Report [WWW15] and the Haskell 98 Library Report [WWW16].

1.2.1. Introduction

Haskell is named after Haskell Brooks Curry [WWW12] who was one of the pioneers of the lambda calculus. Haskell bases on the lambda calculus, a mathematical theory of functions, and not on the Turing machine like imperative programming languages do. In functional programming the programmer defines *what* has to be calculated and not *how* it is calculated. A functional program is a single expression, which is executed by evaluating the expression.

Shortly described, Haskell is a strong typed, lazy, pure functional programming language.

Haskell has a static type system so that all type errors are detected at compile time. This makes Haskell programs very type safe. Its type class system is complex but powerful.

Haskell's evaluation model is called lazy or non-strict, because arguments of functions are only evaluated if they are needed for computations. This leads to a demand-driven evaluation. Expressions are evaluated just enough to get the result. Parts of them may not be evaluated at all. In contrast to imperative languages, program order is not needed.

The language is called pure, because it has no imperative extensions and it eschews all side effects. This leads to the fact that Haskell lacks any loop constructs, because mutable variables do not exist. All iterations have to be expressed by recursions. The lack of side effects allows easy reasoning about the programs.

The above-described qualities make Haskell a modern functional programming language with many strengths over actual object oriented languages like Java or C++.

1.2.2. Functions

Functions are essential for structuring a program. The following example shows a function declaration for adding two numbers.

Example 1-2. Adding two integers

```
add :: Int -> Int -> Int
add a b = a + b
```

The first line is the type signature, it declares the name and type of the function. The function `add` takes two integers as input and returns an integer as a result. The type system of Haskell is so powerful that the type signature can be avoided. However there exist a few exceptions and defining the type signatures makes a program much more maintainable.

The second line gives the definition of the function as an equation. The part before the equal sign names the arguments, the part after the equal sign defines the computation. Using *pattern matching* (see Section 1.2.4) a function can be defined by a series of these equations.

By surrounding the function name of a two-argument function in back-quotes, it can be written between its arguments, called infix.

Example 1-3. Prefix and infix notation

```
add 1 2
1 'add' 2
```

Operators, such as ++ for concatenating two lists, are just infix functions. Instead of writing the function in front of its arguments, it is written between them. Haskell allows the programmer to define his own operators. For defining infix functions, the function name has to be enclosed in parenthesis in the type signature.

Example 1-4. Type signature of an infix function

```
(++) :: [a] -> [a] -> [a]
```

It is also possible to define the associativity and binding power of the self-defined operators. The following example defines ++ as a right-associative operator with a precedence level of 5.

Example 1-5. Define associativity and binding power

```
infixr 5  ++
```

1.2.3. Types

In general terms, a type is a collection of similar objects, such as numbers or characters. Haskell has a very powerful and complex type system. The following sections can just give a small overview. Type classes, abstract data types or infinite lists are not discussed.

1.2.3.1. Build-in types

Like many other programming languages Haskell has the basic build-in types: Char, Int, Bool, Float and String, which is only a synonym for a list of Char.

Lists and tuples are often used data structures for compound data in Haskell and are therefore also build-in values.

Lists combine values of the same type into a single object. The list [1,2,3] is a shorthand for 1:(2:(3:[])). The infix operator : adds its first argument to the front of its second argument, a list. The empty list is expressed by []. The data type list is a *polymorphic type*, see Section 1.2.3.2 .

A tuple combines a predefined number of values of predefined, may be different, types into a single object. Tuples are called records or structures in other programming languages. A person might be represented by a personal id and a name. These two different types can be expressed by a tuple:

Example 1-6. A tuple for persons with ID and name

```
(Int,String) => (42, "Schmidt")
```

1.2.3.2. Polymorphic types

Lists and tuples are the most common examples of generic polymorph data types. A list can contain any type, the only constraint is that all types in one list are the same.

The Prelude, Haskell's standard library, contains lots of polymorphic functions and operators for processing lists and tuples. The head function for example is a list function that returns the head element of any list.

Example 1-7. Get the head of a list

```
head :: [a] -> a
```

The function takes a list with elements of type `a` and returns one element of type `a`, the head. Haskell supports type variables, like the `a` above. They are uncapitalized in contrast to specific types like Char or Int and stand for any type.

1.2.3.3. Type synonyms

Type synonyms are names for commonly used types. They are created using a type declaration. A synonym does not define a new type, it just gives a new name for an existing type. Because synonyms are simply a shorthand, which can always be expanded out, type synonyms cannot be recursive.

As mentioned in the section about basic types, a String is a type synonym for a list of characters.

Example 1-8. Definition of String

```
type String = [Char]
```

Instead of writing ['H','e','l','l','o'] for a string, Haskell defines a shorthand syntax for strings: "Hello".

1.2.3.4. Algebraic types

Algebraic types are used to define more complex types than lists or tuples, e.g. enumerated types or trees. These types can be recursive. Their definition starts with the keyword `data`, followed by the name of the type and the constructors.

The following example demonstrates a recursive data structure for a polymorphic binary tree, which can store any data type in its leafs. The two kinds `Leaf` and `Branch` of the data type `BinTree` are called constructors. They can be used in the pattern matching of function definitions, described in the next section.

Example 1-9. A type for binary trees

```
data BinTree a = Leaf a | Branch (BinTree a) (BinTree a)
```

1.2.4. Pattern Matching

Functions can get different types of input. A powerful way for describing different input types in Haskell is using pattern matching. A function can be multiple defined, each definition having a particular pattern for its input arguments. The first pattern that matches the argument is used for that function call. An underscore is a wildcard and is used where something should match, but where the matched value is not used.

Example 1-10. Calculate the length of a list

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

The function length uses recursion for calculating the length of lists. It calls itself on the right-hand of the second equation. There exist two definitions of the function: one for empty lists and one for lists with content. The pattern `[]` matches the empty list. The pattern `x:xs` matches any list with at least one element. The head element is described by the `x`, the tail of the list by `xs`. The `x` can be replaced by the wildcard, because the head element has not to be accessed directly.

The generic type `[a]` in the function declaration of `length` expresses that `length` is a polymorph function, which can be applied to a list containing elements of any type.

Constructors of algebraic types can also be used for pattern matching. The following function returns a list of all values stored in a binary tree. The infix function `++` concatenates two lists.

Example 1-11. Get all values from Bintree

```
fringe :: Bintree a -> [a]
fringe (Leaf x)       = [x]
fringe (Branch t1 t2) = fringe t1 ++ fringe t2
```

To name a pattern for the use on the right-hand side of an equation, there exists the operator `@`. These patterns are called *as-patterns*. The pattern in front of the `@` always results in a successful match, but the sub-pattern can fail. The following example shows a rewritten fringe function that returns not only the values of the leafs but the leafs itself.

Example 1-12. Get all leafs from Bintree

```
fringe2 :: Bintree a -> [Bintree a]
fringe2 l@(Leaf _)     = [l]
fringe2 (Branch t1 t2) = fringe t1 ++ fringe t2
```

1.2.5. Guards

Guards (`|`) allow boolean tests of arguments passed to a function. The function definition which conditional expression matches first is applied. The keyword `otherwise` matches always. The function `max` uses guards for calculating the maximum of two numbers.

Example 1-13. Calculate the maximum of two numbers

```
max :: Int -> Int -> Int
max x y
    | x <= y     = y
    | otherwise  = x
```

1.2.6. Higher-order Functions

In Haskell functions are first-class citizens. This means that they are themselves simply values. They can be stored in data types or passed as arguments to other functions.

Higher-order functions are functions that take functions as input, return functions as a result or do both. This technique provides a very high abstraction, by embodying a pattern of computation into a function. The following `map` function embodies transformations over lists.

Example 1-14. Apply a function to a list

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

`map` is a polymorphic function that takes a function and a list as arguments. The passed function takes any type `a` and returns any type `b`. By applying `map` to a list of `a`'s, it returns a new list of `b`'s. The generic types `a` and `b` might be different, but do not have to be like the following example demonstrates.

Example 1-15. Increase all numbers in a list

```
map (+1) [1,2,3]  =>  [2,3,4]
```

The infix function + in the example for increasing all numbers in a list is partially applied. Partial application can be done to any function taking two or more arguments. The operator + expects two arguments, one argument is predefined as "1". In this case the function increases every second argument by one.

1.2.7. Modules

Modules are the basis for structuring programs or building general libraries. A module has a name and contains a collection of Haskell definitions. A module may import definitions from other modules and export definitions for the use by other modules.

Example 1-16. Modules

```
module Foo
    ( inc
    , dec)

where import Bar

... definitions of inc and dec
```

The module `Foo` exports the functions `inc` and `dec`. It imports the module `Bar` so that `Foo` can use `Bar`'s exported definitions. Haskell's standard library, the `Prelude`, is implicitly imported.

Chapter 2. Package hdom

This chapter describes the package `hdom`, which defines a generic tree data type `XmlTree` for representing whole XML documents in Haskell. The package provides many functions for processing XML documents represented by this data model.

2.1. Modules

The core public module of the package `hdom` is the module `XmlTree`. It exports all elements from the basic libraries `XmlTreeAccess`, `XmlTreePredicates`, `XmlTreeTypes`, `NTree` and `XmlKeywords`.

The module `XmlTreeTypes` defines data types for representing any XML document in Haskell. The generic tree data type `XmlTree` models XML documents, including all logical units of XML like the DTD subset or document subset. This type is based on the general n-ary tree `NTree` defined in the module `NTree`. The type `NTree` defines trees as a node that has a list of child nodes. Leafs are just nodes with an empty child list. XML documents are composed of elements, comments, DTD declarations, character references or processing instructions. `XmlTree` provides for each logical unit an own node type.

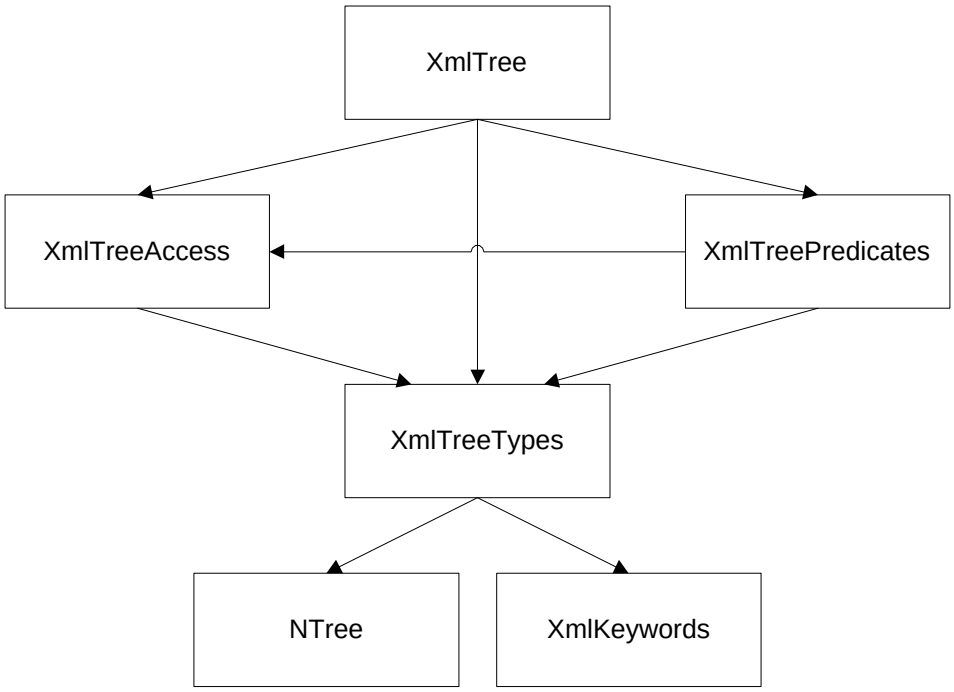
The module `NTree` defines a general n-ary tree structure `NTree` as well as filter functions (see Section 2.3) and combinators (see Section 2.4) for processing this data type. The filters and combinators have been copied and modified from HaXml [WWW21]. In contrast to HaXml the filters have been modified using a more generic approach. The filter functions of HaXml work only for the document subset of XML documents. Because of the generic tree data model `NTree` these functions can be used to process whole XML documents in the Haskell XML Toolbox.

The module `XmlTreeAccess` provides basic filter functions for constructing, editing or selecting parts of the data type `XmlTree`. Furthermore it provides functions for processing the attribute list of `XmlTree` nodes.

The module `XmlTreePredicates` provides basic predicate filter functions. The functions are similar to predicate logic. If the condition is true, they return a list with a node, otherwise they return an empty list.

The module `XmlKeywords` provides constants that are used for representing DTD keywords and attributes in the `XmlTree`.

Figure 2-1. Module hierarchy of XmlTree



Besides the module `XmlTree` and its sub-modules there exist some other public modules for processing the `XmlTree` data structure.

`XmlTreeToString` provides functions for transformation of an `XmlTree` back into a string. The string shows an XML document in XML syntax.

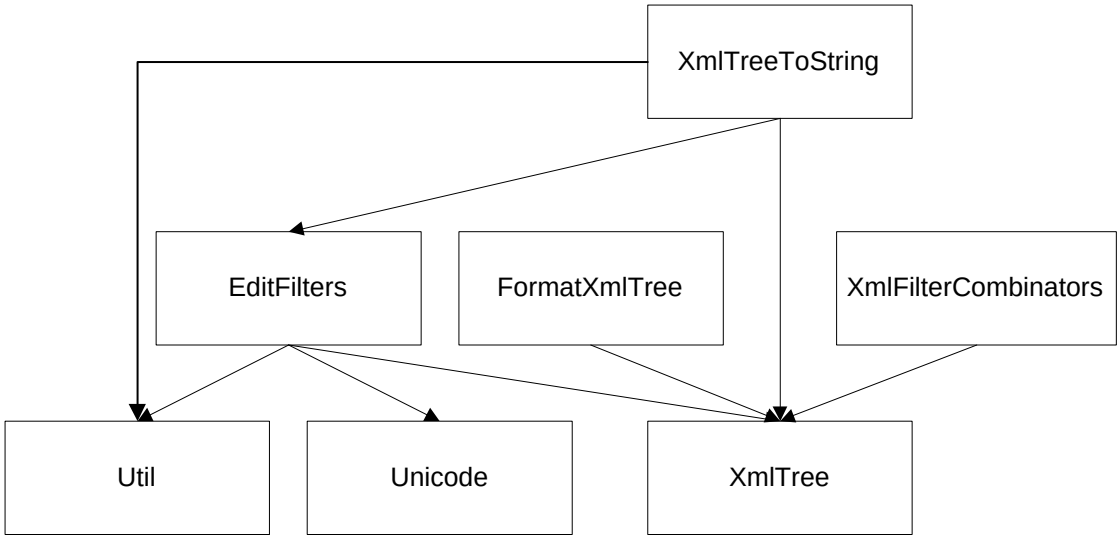
`EditFilters` provides some filters for transforming an `XmlTree`:

- Remove text nodes which contain only white space from the tree.
- Remove comment nodes from the tree.
- Transform special characters like <, >, ", ’ and & of text nodes into character references.
- Manipulate text nodes with customized functions.
- Convert CDATA nodes to text nodes by escaping all special characters.
- Convert character references to normal text.
- Create a canonicalized representation of XML documents [WWW03].

`FormatXmlTree` creates a string with a tree representation of an `XmlTree`. This is useful to see how the data type `XmlTree` represents an XML document.

`XmlFilterCombinators` provides special filter combinators for processing an `XmlTree`. These filters can only operate on the data type `XmlTree` and not on the general n-ary tree data type `NTree`.

Figure 2-2. General modules of package hdom



2.2. Data structures

2.2.1. Core components

XML documents are structured hierarchically and can be represented as trees. Trees can be modeled easily with lists, because trees can be seen as a generalization of lists: nested lists. As shown in the first chapter, Haskell has some nifty facilities for representing and manipulating lists.

XML documents consist of many different logical units, which can be represented by different types. Functional languages like Haskell are well-equipped to process typed tree-structured data by using pattern matching and recursion.

Basically there exist two different approaches for representing XML documents in Haskell. One approach is to use a generic tree structure by which all XML documents can be handled.

Another approach is to use problem specific types that are derived from a specific Document Type Definition. Both approaches have several advantages and disadvantages. They are discussed in depth in the paper "Haskell and XML: Generic Combinators or Type-Based Translation?" by Malcolm Wallace and Colin Runciman [WWW22].

The Haskell XML Toolbox uses a generic data structure for representing whole XML documents. Its data model is more general than the one used in HaXml. The differences are discussed in depth in Section 5.2. A generic data model is the only practical approach for implementing a generic XML parser and a framework for processing any XML documents.

The data type that models a generic tree structure is the data type `NTree`, defined in the module `NTree`. It defines an n-ary tree: a node with a list of children that are also nodes. If a node is a leaf it has an empty child list. The data type `NTrees` is an abbreviation for the node list. Together both types are mutually recursive and form a multi-branch tree structure.

```
data NTree  node = NTree node (NTrees node)
                 deriving (Eq, Ord, Show, Read)

type NTrees node = [NTree node]
```

2.2.2. Data type XmlTree

Building on the very general tree data type `NTree` are defined the types `XmlTree` and `XmlTrees` that are defined in the module `XmlTreeTypes`. They are used to specify a general recursive data type for XML documents. The data, which can be stored in the nodes, must be of type `XNode`.

```
type XmlTree  = NTree  XNode

type XmlTrees = NTrees XNode
```

The algebraic data type `XNode` is used for representing all kinds of XML's logical units. The type is described in detail in the next section.

2.2.2.1. Data type XNode

Before introducing the data type `XNode`, some type synonyms have to be presented which are used in the definitions of `XNode`. The type `TagAttrl` defines the attribute list of an element, it is a list of name-value pairs.

```
type TagAttrl  = [TagAttr]
type TagAttr   = (AttrName, AttrValue)
```

Several type synonyms exist to make the definitions more understandable:

```
type TagName   = String
type AttrName  = String
type AttrValue = String
```

XML documents consist of several different logical units like elements, text data, comments, DTD definitions or entity references. Each kind can be represented by an own type in Haskell so that processing an `XmlTree` can be implemented very efficient by the use of pattern matching. The algebraic data type `XNode` defines the basic nodes and leafs for all kinds of XML's logical units.

Together with the algebraic type `DTDElem`, which defines constructors for the DTD declarations, this data model allows a uniform processing of the whole XML document by filter functions, described in Section 2.3.

```
data XNode =
      XText String
    | XCharRef Int
    | XEntityRef String
    | XCmt String
    | XCdata String
    | XPi TagName TagAttrl
    | XTag TagName TagAttrl
    | XDTD DTDElem TagAttrl
    | XError Int String
    deriving (Eq, Ord, Show, Read)
```

Description of the constructors:

`XText String`

Ordinary text data (leaf)

Note that an `XText` node does not necessarily represent a maximal contiguous sequence of characters. The parser may split text data up into multiple `XText` nodes. The parser produces `XText` nodes from white space occurring before or after tags, too.

There exist special filter functions in the module `EditFilters` for summing up sequences of `XText` nodes into one node and removing `XText` nodes from the tree, which contain white space only.

`XCharRef Int`

Character reference (leaf)

XML syntax: &#nnn; (*nnn* is a hexadecimal or decimal representation of the characters code point in ISO/IEC 10646)

`XEntityRef String`

Entity reference (leaf)

XML syntax: &...;

`XCmt String`

Comment (leaf)

`XCdata String`

CDATA section (leaf)

`XPi TagName TagAttrl`

Processing Instruction (leaf)

`TagName` stores the name of the processing instruction. If name is *xml*, it has the attributes "version", "encoding" and "standalone". Otherwise there is only the attribute "value" which stores a list of the processing instruction attributes.

`XTag TagName TagAttrl`

Element (node or leaf)

Inner node if the element has content, respectively children. Leaf if the element is empty. `TagName` stores the name of the element. The attribute list `TagAttrl` contains all attributes of the element.

`XDTD DTDElem TagAttrl`

DTD element (node or leaf)

The algebraic data type `DTDElem` is used to specify the concrete kind of the element, e.g. element declaration, attribute declaration or entity declaration. The attribute list `TagAttrl` contains almost all information from the DTD declarations (see Section 2.2.2.2).

`XError Int String`

Error (node or leaf)

Not an XML component. Internal extension for representing errors with error level and error message. The error level can be of type: *warning*, *error* or *fatal error*. The error node can have a child list with the nodes where the error occurred.

2.2.2.2. Data type DTDElem

Because a DTD is quite complex, there exists an extra algebraic data type `DTDElem` for representing DTD elements in the `XmlTree`. For each DTD declaration there exists an own type. The nodes store almost all information from DTD declarations in their attribute list. The library `XmlKeywords` provides constants for the names and values of these attributes. Variable contents like the definition of the content model or the names in the value-list of enumerated attribute types are modeled using the node's child list in combination with helper nodes.

```
data DTDElem =
      DOCTYPE
    | ELEMENT
    | CONTENT
    | ATTLIST
    | ENTITY
    | NOTATION
    | NAME
    | PENTITY
    | PEREF
    | CONDSECT
```

```
deriving (Eq, Ord, Show, Read)
```

Description of the constructors:

`DOCTYPE`

The root node of a DTD, has `XDTD` nodes as children.

Attributes:

- name - Name of the root element
- SYSTEM - Reference to external subset (optional)
- PUBLIC - Reference to external subset (optional)

`ELEMENT`

Element declaration. If the element is of type *children* or *mixed*, it has a list of `XDTD CONTENT` nodes as children. These children describe the content model of the element.

Attributes:

- name - Element name
- type - EMPTY | ANY | #PCDATA | children | mixed

`CONTENT`

Not an XML component. Specifies the content model of an element if the element is of type *children* or *mixed*. An `XDTD CONTENT` node has a list of children, which can be of type `XDTD CONTENT` if there is some grouping in the content model, or of type `XDTD NAME` to specify the name of a child element.

Attributes:

- kind - seq | choice
- modifier - "" | ? | * | +

`ATTLIST`

Attribute declaration. If the attribute declaration is of type *NOTATION* or *ENUMERATION*, this node has a list of children of type `XDTD NAME` to specify the enumerated values.

Attributes:

- name - Element name
- value - Attribute name

- kind - #REQUIRED | #IMPLIED | #DEFAULT | #FIXED
- type - CDATA | ID | IDREF | IDREFS | ENTITY | ENTITIES | NMTOKEN | NMTOKENS | NOTATION | ENUMERATION
- default - Default value (optional)

`ENTITY`

General or unparsed entity declaration. If the entity is a general entity, it has a child list with nodes of type `XDTD XCharRef` and `XDTD XText` that specify the replacement text.

Attributes:

- name - Entity name
- SYSTEM - Reference to external file (optional and only for external or unparsed entities)
- NDATA - reference to a notation (optional and only for unparsed entities)

`NOTATION`

Notation declaration.

Attributes:

- name - Notation name
- SYSTEM - Reference to external application

`NAME`

Not an XML component. Used by the attribute declaration `XDTD ATTLIST` to specify the list of values for attributes of type *NOTATION* and *ENUMERATION*.

Used by the node `XDTD CONTENT` to specify element names in content models.

Attributes:

- name - Value of the name

The following node types are not accessible by any application. They are for internal use of the XML parser and handled by it. However it is possible to turn of this processing of the parser, so they are shortly described.

`PENTITY`

Parameter entity declaration.

Attributes:

- name - Name of parameter entity
- value - Value of parameter entity
- SYSTEM - Reference to external file (optional)
- PUBLIC - Reference to external file (optional)

`PEREF`

Parameter entity reference.

Attributes:

- PERef - Name of referenced parameter entity

`CONDSECT`

Node for conditional sections. The child list contains the DTD declarations that are defined in the conditional section.

Attributes:

- type - INCLUDE, IGNORE or parameter entity reference (%...;)

2.2.3. Example

The following example demonstrates how an XML document is transformed into the generic tree structure `XmlTree`.

Example 2-1. Input document

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<!DOCTYPE a [
<!ATTLIST a  att1  CDATA  #IMPLIED>
<!ELEMENT a  (b, c?)>
<!ELEMENT b  EMPTY>
<!ELEMENT c  (#PCDATA)>
]>
```

```
<a att1="test">
    <b/>
    <c>hello world</c>
</a>
```

The following graph shows the `XmlTree` representation of the document after parsing. The predefined general entities *amp*, *lt*, *gt*, *apos* and *quot* are automatically added to the DTD by the XML parser. A dummy node with the name "/" forms the root node of the tree so that the DTD, the document subset, processing instructions and comments can be modeled by one tree.

White space before or after tags is represented by `XText` nodes. There exist filter functions in the module `EditFilters` for removing `XText` nodes containing white space only from the tree if these nodes are not necessary for further processing.

Example 2-2. Input document as XmlTree

```
---XTag "/" [("standalone","yes"),("version","1.0"),("source","example.xml"),
   |         ("encoding","UTF-8")]
   |
   +---XPi "xml" [("version","1.0"),("encoding","UTF-8"),("standalone","yes")]
   |
   +---XText "\n\n"
   |
   +---XDTD DOCTYPE [("name","a")]
   |   |
   |   +---XDTD ENTITY [("name","lt")]
   |   |   |
   |   |   +---XCharRef 38
   |   |   |
   |   |   +---XText "#60;"
   |   |
   |   +---XDTD ENTITY [("name","gt")]
   |   |   |
   |   |   +---XCharRef 62
   .   .
   .   .   (Definitions of the other predefined entities amp, apos and quot)
   .   .
   |   +---XDTD ATTLIST [("name","a"),("value","att1"),("default","42"),
   |   |                 ("kind","#DEFAULT"),("type","CDATA")]
   |   |
   |   +---XDTD ELEMENT [("name","a"),("type","children")]
   |   |   |
   |   |   +---XDTD CONTENT [("modifier",""),("kind","seq")]
   |   |       |
   |   |       +---XDTD NAME [("name","b")]
   |   |       |
   |   |       +---XDTD CONTENT [("modifier","?"),("kind","seq")]
   |   |           |
   |   |           +---XDTD NAME [("name","c")]
   |   |
   |   +---XDTD ELEMENT [("name","b"),("type","EMPTY")]
   |   |
   |   +---XDTD ELEMENT [("name","c"),("type","#PCDATA")]
   |
   +---XText "\n\n"
```

```
   |
   +---XTag "a" [("att1","test")]
   |   |
   |   +---XText "\n    "
   |   |
   |   +---XTag "b" []
   |   |
   |   +---XText "\n    "
   |   |
   |   +---XTag "c" []
   |   |   |
   |   |   +---XText "hello world"
   |   |
   |   +---XText "\n"
   |
   +---XText "\n"
```

The following listing shows the internal representation of the document subset. The root node with the name "a" has a list of attributes and a list of children.

Example 2-3. Internal representation of the document subset

```
NTree (XTag "a" [("att1","test")]) [
    NTree (XText "\n    ") [],
    NTree (XTag "b" []) [],
    NTree (XText "\n    ") [],
    NTree (XTag "c" []) [
        NTree (XText "hello world") []
    ],NTree (XText "\n") []
]
```

2.3. Filter functions

2.3.1. Introduction

Filters are the basic functions for processing the `XmlTree` representation of XML documents. A filter takes a node or a list of nodes and returns some sequence of nodes. The result list might be empty, might contain a single item, or it could contain several items.

The idea of filters was adopted from HaXml [WWW21], but has been modified. In HaXml filters work only on the document subset part of XML documents. The Haskell XML Toolbox uses the generic tree data type `NTree` for modeling XML documents. This generic data model makes it possible to generalize HaXml's filter idea so that filters can process the whole XML document, including the DTD subset or document subset. This generalization allows implementing a very uniform design of XML processing applications by using filters. In fact the whole XML Parser of the Haskell XML Toolbox works internally with filters. The differences between HaXml's approach and the approach of the Haskell XML Toolbox are described in depth in Section 5.2.

`TFilter` and `TSFilter` are filters for the general n-ary tree defined by the data type `NTree`. The function `TFilter` takes a node and returns a list of nodes. `TSFilter` takes a list of nodes and returns a list, too.

```
type TFilter  node = NTree  node -> NTrees node
type TSFilter node = NTrees node -> NTrees node
```

`XmlFilter` and `XmlSFilter` base on these types. They only work on `XNode` data types.

```
type XmlFilter  = TFilter  XNode
type XmlSFilter = TSFilter XNode
```

The filters can be used to select parts of a document, to construct new document parts or to change document parts. They can even be used for checking validity constraints as described in Chapter 4. In this case a filter returns an empty list if the document is valid or a list with errors.

Filters can sometimes be thought of as predicates. In this case they are used for deciding whether or not to keep its input. The functional approach differs from predicate logic. If the predicate is false, an empty list is returned. If the predicate is true, a list with the passed element is returned.

All filters share the same basic type so that combining them with the help of combinators, described in Section 2.4, is possible. With this approach defining complex filters on the basis of easier ones is possible.

The following list describes the basic filter functions for processing XML documents represented as an `XmlTree`. Some functions are higher-order functions and return a filter function as a result. The arguments of these functions are used to construct parameterized filters. This is useful for example for constructing filters that should be used to return nodes with a certain property.

2.3.2. Filters from module NTree

Simple filters

`none :: TFilter node`

Takes any node, returns always an empty list. Algebraically zero.

`this :: TFilter node`

Takes any node, returns always a list of the passed node. Algebraically unit.

Selection filters

`isOfNode :: (node -> Bool) -> TFilter node`

Takes a predicate functions and returns a filter. The filter returns a list with passed node if the predicate function is true for the node, otherwise it returns an empty list.

`isNode :: Eq node => node -> TFilter node`

The same as `isOfNode`. Instead of a predicate function a reference node is taken.

Filters for modifying nodes

`mkNTree :: NTree node -> TFilter node`

Takes a node and returns a filter. The filter returns always a list with this node and ignores the passed one.

`replaceNode :: node -> TFilter node`

Takes a node and returns a filter. The filter replaces a passed node with the initialized one. The children of the passed node are added to the new node.

`replaceChildren :: NTrees node -> TFilter node`

Like `replaceNode` except that in this case the children are replaced by an initialized list. The passed node itself is not modified.

`modifyNode :: (node -> Maybe node) -> TFilter node`

Takes a function for modifying nodes and returns a filter. The function `(node -> Maybe node)` is applied to the passed node, the children are not modified.

`modifyNode0 :: (node -> node) -> TFilter node`

Like `modifyNode0` except that the type of the modification function `(node -> node)` is different.

`modifyChildren :: TSFilter node -> TFilter node`

Takes a filter that processes lists of nodes and returns a new filter. The new filter applies the filter `TSFilter node` to the child list of a passed node. The node itself is not modified.

2.3.3. Filters from module XmlTreeAccess

Predicate filters

`isTag :: TagName -> XmlFilter`

Takes a `TagName` and returns a filter. The filter returns a list with the passed node if its name equals `TagName`, otherwise an empty list is returned.

`isOfTag :: (TagName -> Bool) -> XmlFilter`

Takes a predicate function `(TagName -> Bool)` and returns a filter. The filter applies the predicate function to the name of a passed node. If the predicate function is true, the filter returns a list with the node. Otherwise an empty list is returned.

`attrHasValue :: AttrName -> (AttrValue -> Bool) -> XmlFilter`

Constructs a predicate filter for attributes which value meets a predicate function. The constructed filter returns a list with the passed node if the node has an attribute with name `AttrName` and its value matches the predicate function `(AttrValue -> Bool)`. Otherwise an empty list is returned.

Lots of further predicate functions are provided by the module `XmlTreePredicates`: `isXCdata`, `isXCharRef`, `isXCmt`, `isXDTD`, `isXEntityRef`, `isXError`, `isXNoError`, `isXPi`, `isXTag`, `isXText`, etc. These filters are used for identifying special types of nodes.

Construction filters

`mkXTag :: TagName -> TagAttrl -> XmlTrees -> XmlFilter`

The created filter constructs an `XTag` node with the name `TagName`, an attribute list `TagAttrl` and a list of children. The passed node is ignored by the filter.

`mkXText :: String -> XmlFilter`

The created filter constructs an `XText` node with text data. The passed node is ignored by the filter. There exists a shortcut function `txt` that does the same.

`mkXCharRef :: Int -> XmlFilter`

The created filter constructs an `XCharRef` node with a reference number to a character. The passed node is ignored by the filter.

`mkXEntityRef :: String -> XmlFilter`

The created filter constructs an `XEntityRef` node with an entity reference. The passed node is ignored by the filter.

`mkXCmt :: String -> XmlFilter`

The created filter constructs an `XCmt` node with text data. The passed node is ignored by the filter. There exists a shortcut function `cmt` that does the same.

`mkXDTD :: DTDElem -> TagAttrl -> XmlTrees -> XmlFilter`

The created filter constructs an `XDTD` node. The type of the node is specified by the algebraic data type `DTDElem`. The node has attributes and a list of children. The passed node is ignored by the filter.

`mkXPi :: String -> TagAttrl -> XmlFilter`

The created filter constructs an `XPi` node with a name and attributes. The passed node is ignored by the filter.

`mkXCdata :: String -> XmlFilter`

The created filter constructs an `XCdata` node with text data. The passed node is ignored by the filter.

`mkXError :: Int -> String -> XmlFilter`

The created filter constructs an `XError` node with an error level and an error message. The passed node is stored in the child list of this error node, so that the location where the error occurred can be preserved. The shortcut functions `warn`, `err` and `fatal` of type `String -> XmlFilter` can be used to create specific error nodes.

`mkXElem :: TagName -> TagAttrl -> [XmlFilter] -> XmlFilter`

The created filter constructs an `XTag` node with the name `TagName` and the attribute list `TagAttrl`. Its child list is constructed by applying the filter list `[XmlFilter]` to the passed node. There exists a shortcut function `tag` that does the same.

`mkXSElem :: TagName -> [XmlFilter] -> XmlFilter`

The created filter constructs a simple `XTag` node. It works like `mkXElem` except that no attribute list is created. There exists a shortcut function `stag` that does the same.

`mkXEElem :: TagName -> XmlFilter`

The created filter constructs an empty `XTag` node. It works like `mkXSElem` except that no child list is created. There exists a shortcut function `etag` that does the same.

Selection filters

`getXTagName :: XmlFilter`

If the passed node is of type `XTag`, a list with an `XText` node is returned. This node contains the name of the element. Otherwise an empty list is returned.

`getXTagAttr :: AttrName -> XmlFilter`

If the passed node is of type `XTag` and there exists an attribute with the name `AttrName`, a list with an `XText` node is returned. This node contains the value of the attribute. Otherwise an empty list is returned.

`getXDTDAttr :: AttrName -> XmlFilter`

The same as `getXTagAttr` except that it works on `XDTD` nodes.

`getXText :: XmlFilter`

If the passed node is of type `XText`, a list with an `XText` node is returned. This node contains text data. Otherwise an empty list is returned.

`getXCmt :: XmlFilter`

If the passed node is of type `XCmt`, a list with an `XText` node is returned. This node contains the text data of the comment. Otherwise an empty list is returned.

`getXPiName :: XmlFilter`

If the passed node is of type `XPi`, a list with an `XText` node is returned. This node contains the name of the processing instruction. Otherwise an empty list is returned.

`getXCdata :: XmlFilter`

If the passed node is of type `XCdata`, a list with an `XText` node is returned. This node contains the text data of the element. Otherwise an empty list is returned.

`getXError :: XmlFilter`

If the passed node is of type `XError`, a list with an `XText` node is returned. This node contains the error message. Otherwise an empty list is returned.

Substitution filters

`replaceTagName :: TagName -> XmlFilter`

Constructed filter replaces the name of an `XTag` or `XPi` node by the `TagName` and returns a list with the modified node.

`replaceAttrl :: TagAttrl -> XmlFilter`

Constructed filter replaces the attribute list of an `XTag`, `XDTD` or `XPi` node by the `TagAttrl` and returns a list with the modified node.

`modifyTagName :: (TagName -> TagName) -> XmlFilter`

Constructed filter modifies the name of an `XTag` or `XPi` node by applying the function `(TagName -> TagName)` to the name. The filter returns a list with the modified node.

`modifyAttrl :: (TagAttrl -> TagAttrl) -> XmlFilter`

Constructed filter modifies the attribute list of an `XTag`, `XDTD` or `XPi` node by applying the function `(TagAttrl -> TagAttrl)` to the attribute list. The filter returns a list with the modified node.

`modifyAttr :: AttrName -> AttrValue -> XmlFilter`

Constructed filter changes the attribute value of the attribute which name equals `AttrName` to `AttrValue` and returns a list with the modified node.

2.4. Filter combinators

2.4.1. Introduction

Functional programming languages like Haskell allow the use of higher-order functions, which take some functions as input, return a function as a result, or both. Filter combinators are higher-order functions, or operators, for combining several filters (see Section 2.3). By combining simple filters, more complex functions can be created. Because all filters share the same type, it is possible that any filter can be composed with any other. All filters can be mixed and matched freely, because all functions in Haskell are side-effect free.

The idea of filter combinators was adapted and extended from HaXml [WWW21]. In their paper "Haskell and XML: Generic Combinators or Type-Based Translation?" [WWW22] Malcom Wallace and Colin Runciman describe various algebraic laws for their combinators.

The use of combinators makes it possible to hide details of programming over data structures. All details of data structure manipulation are hidden in combinator functions. In effect, the combinators define problem specific control structures. With this approach it is possible to reach a form of expression that is natural for the problem itself.

Conceptually, combinators are much like Unix pipes in that they allow building up more complex computational sequences by flexibly arranging highly specialized tools [WWW24]. The equivalent to Unix pipes it the "Irish composition" combinator, represented by the infix operator "`o`". This combinator applies one filter to the results of another one. This is similar to a pipe, which passes the output of one program as the input to another one.

The combinator library provides all functions that are necessary for traversing and processing XML documents represented as an `XmlTree`. Haskell allows the definition of own infix operator symbols. Some combinators are defined as infix operators where it seemed more natural.

2.4.2. List of combinators

Basic filter combinators

`o :: (a -> [b]) -> (c -> [a]) -> c -> [b]`

Irish composition. Sequential composition of two filters. The left filter is applied to the result of the right filter. (conjunction)

`(+++) :: (a -> [b]) -> (a -> [b]) -> (a -> [b])`

Binary parallel composition, the function unifies the results of two filters sequential. Each filter uses a copy of state. (union)

`cat :: [a -> [b]] -> (a -> [b])`

Concatenates the results of all filters in a list, a list version of union `+++`. The combinator sticks lots of filters together.

`($$) :: (a -> [b]) -> [a] -> [b]`

Applies a filter to a list.

`processChildren :: TFilter node -> TFilter node`

Applies a filter to the child list of a node.

Choice combinators

`orElse :: (a -> [b]) -> (a -> [b]) -> (a -> [b])`

Directional choice. Second filter is only applied if the first one produces an empty list.

`(?>) :: (a -> [b]) -> ThenElse (a -> [b]) -> (a -> [b])`

In combination with the type `ThenElse a = a :> a` this combinator models an expression that resembles the conditional expression "`p ? f : g`" of C: "`p ?> f :> g`". If the predicate filter `p` is true, the filter `f` is applied otherwise `g` is applied.

`when :: TFilter node -> TFilter node -> TFilter nod`

First filter is only applied to the passed node, if the second filter produces a list with contents, otherwise a list with the passed node is returned. The second filter is typically a predicate filter.

Definition: `f ‘when‘ g = g ?> f :> this`

`whenNot :: TFilter node -> TFilter node -> TFilter node`

First filter is only applied to the passed node, if the second filter produces an empty list, otherwise a list of the passed node is returned. The second filter is typically a predicate filter.

Definition: `f ‘whenNot‘ g = g ?> this :> f`

`guards :: TFilter node -> TFilter node -> TFilter node`

First filter is only applied to the passed node, if the second filter produces a list with contents, otherwise an empty list is returned. The second filter is typically a predicate filter.

Definition: `g ‘guards‘ f = g ?> f :> none`

`containing :: (a -> [b]) -> (b -> [c]) -> a -> [b]`

Pruning: Keep only those nodes for which the second filter produces a list with contents and apply the first filter to these nodes.

Definition: `f ‘containing‘ g = filter (not . null . g) . f`

`notContaining :: (a -> [b]) -> (b -> [c]) -> a -> [b]`

Pruning: Keep only those nodes for which the second filter produces an empty list and apply the first filter to these nodes.

Definition: `f ‘notContaining‘ g = filter (null . g) . f`

`(/>) :: TFilter node -> TFilter node -> TFilter node`

Interior search, pronounced *"slash"*. First filter is applied to a node, the children of the result are taken and the second filter is applied to them, these children are returned. (meaning g inside f)

Definition: `f /> g = g ‘o‘ getChildren ‘o‘ f`

`(</) :: TFilter node -> TFilter node -> TFilter node`

Exterior search, pronounced *"outside"*. First filter is applied to a node, the second to its children. If both filters return a result, the node is returned. (meaning f containing g)

Definition: `f </ g = f ‘containing‘ (g ‘o‘ getChildren)`

Recursive search combinators

`deep :: TFilter node -> TFilter node`

Filter is applied to each node of the tree. If the filter returns a result, processing is stopped. If not, the filter is applied to the next node of the tree. (top down traversal)

`deepest :: TFilter node -> TFilter node`

See `deep`, but bottom up traversal.

`multi :: TFilter node -> TFilter node`

Returns all successful applications of the filter to the whole tree.

Recursive transformation combinators

`applyBottomUp :: TFilter node -> TFilter node`

Constructs a new tree by applying the passed filter to each node of the tree. The tree is traversed bottom-up.

`applyTopDown :: TFilter node -> TFilter node`

Constructs a new tree by applying the passed filter to each node of the tree. The tree is traversed top-down.

`applyBottomUpIfNot :: TFilter node -> TFilter node -> TFilter node`

Constructs a new tree with a guarded top down transformation. The first filter is only applied to those nodes for which the second filter produces a result.

Monadic composition

`liftM :: Monad m => (a -> [b]) -> a -> m [b]`

Lift a normal filter to a monadic filter so that it can be used in monads.

`(.<) :: Monad m => (b -> m [c]) -> (a -> m[b]) -> (a -> m[c])`

Monadic Irish composition. Sequential composition of two filters. The first filter is applied to the result of the second filter (conjunction). Monadic version of the basic filter `o`.

`($$<) :: Monad m => (a -> m [b]) -> [a] -> m [b]`

Apply a monadic filter to a list. Monadic version of the basic filter combinator `cat`.

Special Filter combinators for XmlTree

`whenOk :: XmlFilter -> XmlFilter ->XmlFilter`

Applies the first filter to a node and checks the result for errors. In case of an error it stops processing and returns the list with the `XError` node. Otherwise processing is continued and the second filter is applied to the result.

`whenOkM :: Monad m => (a -> XmlTrees) -> (XmlTrees -> m XmlTrees) -> a -> m XmlTrees`

Monadic version of `whenOk`.

2.4.3. Binding power and associativity

Some combinators like the basic filter combinators and choice combinators can be expressed more naturally by operators. In Haskell completely new operators can be invented by using infix versions of functions and defining precedences and associativities for them.

The combinator operators are listed in following tables in decreasing order of their binding power. Combinators defined as prefix functions have to be written in back-quotes to be used as operators.

Table 2-1. Binding power and associativity of functional combinators

Precedence level	Operator	Associativity
6 (high)	`‘containing‘`, `‘notContaining‘`	left
5	`‘o‘`, `+++`	right
5	`/>`, `</`, `‘orElse‘`	left
4	`‘when‘`, `‘whenNot‘`, `‘guards‘`, `‘whenOk‘`	right
3	`?>`, `:>`	right
0 (low)	`$$`	right

Table 2-2. Binding power and associativity of monadic combinators

Precedence level	Operator	Associativity
5 (high)	`.<`	right
4	`‘whenOkM‘`	right
0 (low)	`$$<`	right

2.5. Examples for filters and filter combinators

After discussing filters and filter combinators in depth, this section shows the use of these functions in real life examples. Because the filters work on the generic tree data type `XmlTree`, which represents whole XML documents, these functions form the basic uniform design of XML processing applications. The whole XML Parser of the Haskell XML Toolbox works internally with filters.

2.5.1. Removing comments

This example describes in depth the use of filters and combinators for removing comments from an `XmlTree`. It illustrates, how a complex filter can be constructed by combining very simple ones.

Removing comments from an `XmlTree` is done by transforming the tree with a special filter. This filter returns an empty list for comment nodes, all other nodes are simply returned. The result is a new tree without comment nodes.

To implement this filter, a predicate function is needed first, which detects comment nodes. Building on this filter a more complex one will be constructed that returns an empty list for comment nodes, but returns all other nodes. The final filter will apply this filter to the whole tree and create a new tree from its results.

The function `isXCmt` is a predicate filter that detects comment nodes. It takes a node and checks if the node is of type `XCmt`. If the node is a comment, a list with this node is returned, otherwise the

result is an empty list.

```
isXCmt              :: XmlFilter
isXCmt              = isOfNode isXCmtNode
```

The predicate filter `isXCmt` itself bases on the basic selection filter `isOfNode`, which is exported by the module `XmlTree`, and the predicate function `isXCmtNode`. The selection filter `isOfNode` takes the predicate function as a parameter and returns a list with the passed node if the predicate function returns true for this node. Otherwise an empty list is returned. The predicate function `isXCmtNode` uses pattern matching to figure out if a node is of type `XCmt`.

```
isOfNode            :: (node -> Bool) -> TFilter node
isOfNode p t@(NTree n _)
    | p n       = [t]
    | otherwise = []

isXCmtNode          :: XNode -> Bool
isXCmtNode (XCmt _) = True
isXCmtNode _        = False
```

The filter `removeComment` takes a node and returns an empty list if the node is a comment, otherwise a list with the passed node is returned. The filter `removeComment` combines the simple filter `none` and the above described filter `isXCmt` with the combinator `when`. If the predicate filter `isXCmt` is true, `none` returns an empty list. Otherwise a list with the node is returned.

```
removeComment :: XmlFilter
removeComment
    = none `when` isXCmt
```

The main filter `removeAllComment` is a filter for removing all comments from an `XmlTree`. It uses the recursive transformation filter `applyBottomUp`, which applies the filter `removeComment` to the whole tree. The result is a new tree where all comments have been removed.

```
removeAllComment :: XmlFilter
removeAllComment
    = applyBottomUp removeComment
```

This example shows very clearly how complex filters can be constructed out of simple ones. The functions `isOfNode`, `none`, `when` and `applyBottomUp` are already exported by the module `XmlTree`. They define own control structures and hide processing of the `XmlTree` from the programmer. The programmer does not have to worry about how to apply a filter to the whole tree and how to transform it. The use of predefined combinators and filters makes it possible to program on a very high abstraction level.

2.5.2. Merging internal and external DTD subset

After discussing the simple example for removing comments, this example will show a much more complex use of filters. Having defined combinators and basic filters already, it shows how they can usefully be combined into a complex function for merging the internal and external DTD subset. The

XML parser of the Haskell XML Toolbox uses this function internally. The filter is part of the module `DTDProcessing`.

A validating XML parser must merge the internal and external DTD subset. The document type declaration can point to an external subset containing declarations, it can contain the declarations directly in an internal subset, or can do both. If both the external and internal subsets are used, the internal subset must occur before the external subset. This has the effect that entity and attribute-list declarations in the internal subset take precedence over those in the external subset [WWW01].

The function `mergeDTDs` takes two lists with the declarations of the internal and external subset as input and returns both subsets merged. Because the internal subset dominates over the external one, its declarations can be prepended before the filtered result list of the external subset. The operator `++` is a standard Haskell function for concatenating two lists. The expression `mergeDTDentry dtdInt` creates a complex filter function that filters declarations from the external subset, which have been already declared in the internal subset. This complex filter is applied by the combinator `$$` to the whole declaration list of the external DTD subset.

```
mergeDTDs :: XmlTrees -> XmlTrees -> XmlTrees
mergeDTDs dtdInt dtdExt
    = dtdInt ++ (mergeDTDentry dtdInt $$ dtdExt)
```

The complex filter `mergeDTDentry` is initialized with the internal subset. By applying this filter to each node of the external subset, it filters all entries that are already declared in the internal subset. It uses the same semantic like the example for removing comments. If a declaration occurs in both subsets, the filter returns an empty list, otherwise it returns a list with the element from the external subset. This behavior is defined by the expression: `none ‘when‘ found`.

Internally the filter `mergeDTDentry` bases on the filter `filterDTDNode`, which returns a node that occurs in both subsets. This filter is initialized with one node from the internal subset and checks if a node from the external subset equals this node. Because there usually exists more than one declaration in the internal subset, a list of this filter function is constructed, by applying `filterDTDNode` with the list transformation function `map` to all nodes of the internal subset. The result is a list of filter functions. This list of filters is later applied to a node of the external subset by the combinator `cat`.

```
mergeDTDentry :: XmlTrees -> XmlFilter
mergeDTDentry dtd
    = none ‘when‘ found
      where
      filterList = map filterDTDNode dtd  -- construct the list of filters
      found      = cat filterList         -- concatenate the filters (set union)
```

The function `filterDTDNode` constructs a filter which returns a list with a node that occurs in both subsets. The filter is initialized with a node from the internal subset.

```
filterDTDNode :: XmlTree -> XmlFilter
```

If the nodes from the internal and external subsets are both of type `ELEMENT`, `NOTATION` or `ENTITY`, the filter checks if the values of their attributes "*name*" are equal. If this is the case, a list with the node is returned, otherwise an empty list is returned.

```
filterDTDNode (NTree (XDTD dtdElem al) _)
    | dtdElem ‘elem‘ [ELEMENT, NOTATION, ENTITY]
        = filterElement
          where
          filterElement n@(NTree (XDTD dtdElem’ al’) _cl’)
              | dtdElem == dtdElem’ &&
                getAttrValue a_name al’ == getAttrValue a_name al
                  = [n]
              | otherwise  = []
          filterElement _  = []
```

If the nodes from the internal and external subset are of type `ATTLIST`, the filter has to check if the values of the attributes for the element name and attribute name are equal. If this is the case, the declarations are equal and a list with this node is returned, otherwise an empty list is returned.

```
filterDTDNode (NTree (XDTD ATTLIST al) _)
    = filterAttlist
      where
      filterAttlist n@(NTree (XDTD ATTLIST al’) _cl’)
          | getAttrValue a_name  al’ == getAttrValue a_name  al &&
            getAttrValue a_value al’ == getAttrValue a_value al
              = [n]
      filterAttlist _ = []
```

For all other types a filter is constructed that returns always an empty list.

```
filterDTDNode _ =  none
```

This example might be a little bit confusing for people being not familiar with Haskell, but it demonstrates how even very complex tasks like merging the internal and external DTD subset can be implemented with filter functions and combinators. In this case a list of parameterized filter functions is constructed at runtime, which is applied to each node of the external DTD subset.

2.6. Access functions

Besides filters and combinators there exist some access functions for processing the attribute list `TagAttrl` of nodes.

Functions for processing the attribute list

`getAttrValue :: Eq a => a -> [(a,b)] -> Maybe b`

Looks up the value for a certain attribute. If the attribute does not exist, the data type `Nothing` is returned.

`getAttrValue1 :: Eq a => a -> [(a, String)] -> String`

Looks up the value for a certain attribute. If the attribute does not exist, an empty string is returned.

`hasAttrValue :: Eq a => a -> [(a,b)] -> Bool`

Returns true if there is an attribute with the searched name in the attribute list.

`addAttrValue :: Eq a => a -> b -> [(a,b)] -> [(a,b)]`

Adds an attribute, the name and value, to the list.

`addAttrValues :: Eq a => [(a,b)] -> [(a,b)] -> [(a,b)]`

Adds a list of name-value pairs to the attribute list.

`changeAttrValue :: Eq a => a -> b -> [(a,b)] -> [(a,b)]`

Changes the value of a certain attribute.

`delAttrValue :: Eq a => a -> [(a,b)] -> [(a,b)]`

Deletes an attribute from the list.

`delAttrValues :: Eq a => [a] -> [(a,b)] -> [(a,b)]`

Deletes a list of attributes from the attribute list.

The following example shows the use of access functions for adding a new attribute with the name "*att2*" and the value "*val2*" to the attribute list of an `XTag` node. The type of the node is detected by pattern-matching. If the node is of another type, it is simply returned.

Example 2-4. Adding a new attribute to an XTag node.

```
addAttribute :: XmlFilter
addAttribute n@(NTree (XTag name al) cs)
    = replaceAttrl newAttList n
      where
      newAttList = addAttrValue "att2" "val2" al

addAttribute n = [n]
```

2.7. State-I/O monad from module XmlState

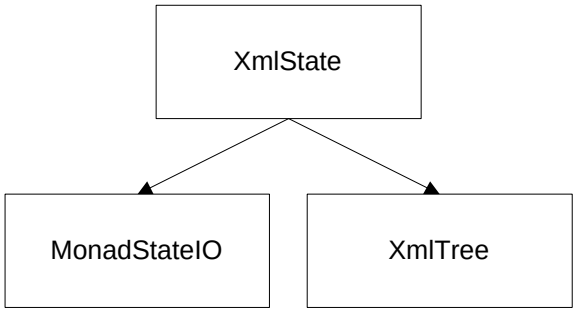
The module `XmlState` provides a monad for an internal state and I/O commands. The concept of a monad comes from category theory. Monads are used for programming in a functional language using an imperative style. A monad can encapsulate computations such as I/O, manipulation of state or exceptions. For further information about monads, the section "Using Monads" of the Haskell Bookshelf [WWW17] is a good starting place.

`XmlState` is used for various parts of the XML parser, e.g. when building the parse tree or substituting entities. The internal state of the monad from `XmlState` consists of two parts, the user state and the system state. The user state is a type parameter, the system state is a list of name-value pairs. If the user state is not needed, the type parameter can be instantiated with `()`.

The monad provides trace functions combined with trace levels to output marked computation steps. Error reporting functions are also located in this module. The error messages are printed to *stderr* and the maximum error level is stored in the system state.

Furthermore there are types for `XmlFilter` functions working on this monad, functions for manipulating the state components and for lifting I/O commands and `XmlFilters` to monad filters.

Figure 2-3. Modules of XmlState



The internal state consists of a system and a user state.

```
data XmlState state       = XmlState { sysState  :: SysState
                                     , userState :: state
                                     }
```

The system state consists of a list of name-value pairs of type String.

```
type SysState             = TagAttrl
```

The monad type for commands. It is an instance of `StateIO` from the general module `MonadStateIO`.

```
type XState state res     = MonadStateIO.StateIO (XmlState state) res
```

The `XmlFilter` type for filters working on a state.

```
type XmlStateFilter state = XmlTree -> XState state XmlTrees
```

Functions for executing XState commands

`run0 :: XmlState state -> XState state res -> IO (res, XmlState state)`

Executes an XState command with an initial state.

`run :: state -> XState state res -> IO res`

Executes an XState command with an initial user state.

`run' :: XState () res -> IO res`

Executes an XState command in the I/O monad.

The following example shows the use of the State-I/O monad. All computations of `processXmlN` take place in the State-I/O monad. The functions `getXmlContents`, `parseXmlDoc` and `putXmlTree` are `XmlStateFilters`. The first two filters do some computations where errors might occur. These `XmlStateFilters` can output the errors and set an error level in the State-I/O monad. The value of the error level is tested before the constructed parse tree is returned. The `XmlStateFilters` `putXmlTree` just outputs the constructed parse tree.

Example 2-5. Using the monad from XmlState

```
processXmlN :: XmlTree -> IO [XmlTree]
processXmlN t0
    = run' $ do
              setSysState (selXTagAttrl . getNode $ t0)
              setTraceLevel 0
              t1 <- getXmlContents $ t0
              t2 <- parseXmlDoc    $$< t1
              putXmlTree t2
              el <- getErrorLevel
              return ( if el == 0
                       then t2
                       else [] )
```

The functions `setSysState`, `setTraceLevel` and `getErrorLevel` are functions for accessing and manipulating the state of the monad. The monad comes with lots of more access functions.

Chapter 3. Package hparser

The following chapter describes the package `hparser`. It contains all modules for parsing XML files and checking the well-formedness constraints of XML. Validation, normalizing attribute values and adding default values is done by the separate package `hvalidator`.

The parser works internally with filters for reading XML data and processing the constructed `XmlTree`. The whole parser works with filters of type `XmlStateFilter` from the module `XmlState` so that trace-output of the computations can be generated, errors can be reported and some state information can be saved where it is necessary.

3.1. Overview

The package `hparser` provides five public modules for implementing an XML parser, which builds the generic tree data structure `XmlTree` from XML documents.

Modules

`HdomParser`

Basic parser for building the generic tree data structure `XmlTree`. Bases on `XmlParser`, `XmlInput` and `DTDProcessing`.

`XmlParser`

Parses XML files. Bases on the free monadic parser combinator library Parsec [WWW29].

`XmlInput`

Reads XML files, handles different character encodings.

`DTDProcessing`

Processes the DTD. Substitution of general and parameter entities, merging of the internal and external DTD subset.

`XmlOutput`

Prints the parsed data by using the `XmlStateFilter` defined in module `XmlState`.

3.2. Module HdomParser

The module `HdomParser` provides functions for parsing XML files and building the generic tree data structure `XmlTree` from these documents. The whole parsing process takes place in the State-I/O monad of the module `XmlState`, so that well-formedness errors can be reported and different computations can be traced by outputting their results.

Parse functions

`parseDoc :: String -> IO [XmlTree]`

Parses the file specified by the first parameter.

`parseXmlFile :: IO [XmlTree]`

Parses an XML file specified by a command line argument:

- --source "source file" - XML file to parse
- --encoding "encoding" - Encoding scheme used in the file (optional)

The following example shows how an XML parser is constructed in the module `HdomParser`. The whole parsing process takes place in the State-I/O monad defined in the module `XmlState`. All computations are of type `XmlStateFilter`.

```
processXmlN :: Int -> XmlTree -> IO [XmlTree]
processXmlN n t0
    = run' $ do
              setSysState (selXTagAttrl . getNode $ t0)
              setTraceLevel n
              t1 <- getXmlContents $ t0
              t2 <- parseXmlDoc    $$< t1
              t3 <- liftM transfAllCharRef $$< t2
              t4 <- processDTD     $$< t3
              t5 <- processGeneralEntities $$< t4
              el <- getErrorLevel
              return ( if el == 0
                       then t5
                       else [] )
```

Actions during parsing

`getXmlContents :: XmlStateFilter a`

Returns a filter for reading the XML file. The filename is retrieved from the attribute with the name "*source*" which must be part of the initial node *t0*.

`parseXmlDoc :: XmlStateFilter a`

Parses the XML file and builds the `XmlTree`.

`transfAllCharRef :: XmlFilter`

The `XmlFilter` `transfAllCharRef` has to be lifted to an `XmlStateFilter`. The filter substitutes character references by their characters.

`processDTD :: XmlStateFilter a`

Substitutes parameter entities, adds include sections and removes exclude sections of DTDs, merges internal and external DTD subsets.

`processGeneralEntities :: XmlStateFilter a`

Substitutes general entities.

```
getErrorLevel :: XState state Int
```

If an error occurred during applying the `XmlStateFilters`, `processXmlN` returns an empty list, otherwise it returns the constructed `XmlTree`.

3.3. Module XmlParser

The module `XmlParser` provides the parse functions for parsing XML files and building an `XmlTree`. The parser bases on Parsec [WWW29], a free monadic parser combinator library for Haskell and does not need any look-ahead. The lexer and the parser are not separated. A feature of this parser is that nearly all parse functions can be implemented as it is defined by the productions in the XML 1.0 specification [WWW01].

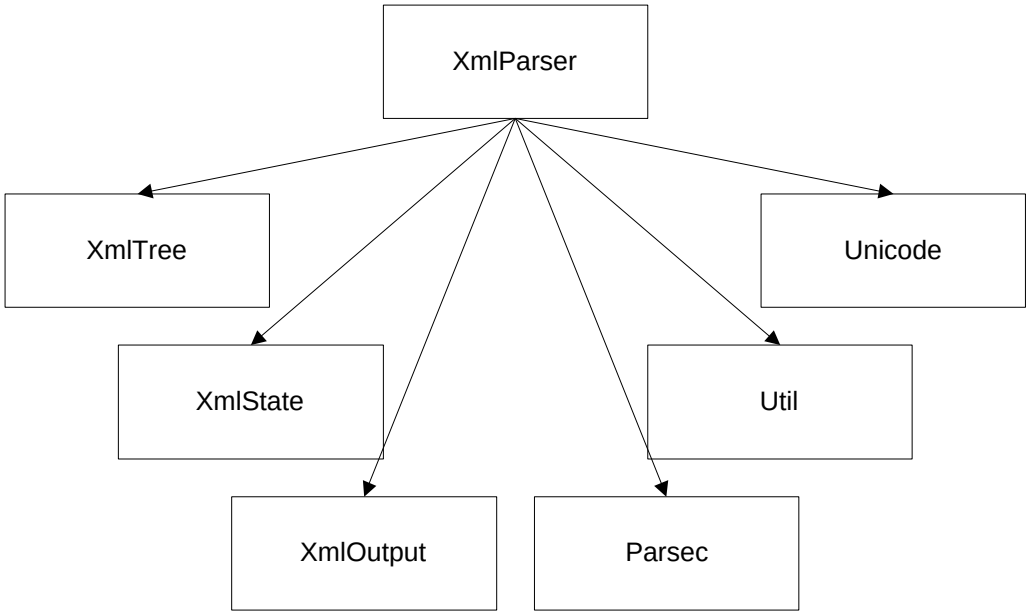
Like filters are composed with filter combinators in the Haskell XML Toolbox, parsing is usually done with parser combinators in Haskell. Simple parser functions are combined with these higher-order functions to complex ones. The parser combinators are control structures that represent the operators used it the productions of syntax definitions.

An XML parser has to deal with different character sets. The parser of the Haskell XML Toolbox works internally with Unicode (UTF-8) encoding. The module `Unicode` provides several conversion functions for different character sets.

Supported character sets by the XmlParser:

- UTF-8
- ISO-8859-1
- US-ASCII
- ISO-10646-UCS-2
- UTF-16
- UTF-16BE
- UTF-16LE

Figure 3-1. Modules of XmlParser



3.4. Module XmlInput

The module XmlInput provides input functions implemented as filters of type `XmlStateFilter`. The main function is `getXmlContents`, which returns an `XmlStateFilter` for reading the content. This filter expects as input a node of type `NTree XTag` with several arguments stored in its attribute list. The child list of this node should be empty, because the parsed document will be stored there.

Protocols like the *http* or *file* protocol are not supported at the moment. Unfortunately there does not seem to exist any Haskell library supporting theses protocols. Perhaps later versions of the parser will use the `curl` command.

Arguments of getXmlContents

Keyword a_source ("source") from module `XmlKeywords`

Specifies input file (or later URL)

Keyword a_encoding ("encoding") from module `XmlKeywords`

Specifies encoding scheme (optional). If no encoding is specified, the parser tries to guess the encoding.

3.5. Module DTDProcessing

The module `DTDProcessing` provides functions for parameter entity substitution, general entity substitution and merging the internal and the external DTD subset into one internal part. Parameter entity substitution should be done before merging the subsets of a DTD. The function `processDTD` ensures this. Substitution of general entities is done by `processGeneralEntities`. The filters are monadic functions that use the State-I/O monad from the module `XmlState`.

3.6. Module XmlOutput

`XmlOutput` provides output functions implemented as `XmlStateFilters`. These filters are very useful for printing out the `XmlTree` during computations while parsing, e.g. printing out the tree before and after entity processing. The whole parsing process takes place in the State-I/O monad from `XmlState`. If a trace level different from zero is set in this monad, these functions are used to show the computation steps.

`putXmlSource :: XmlStateFilter a`

Prints the `XmlTree` in XML representation.

`putXmlTree :: XmlStateFilter a`

Prints the `XmlTree` in tree representation.

Chapter 4. Package hvalidator

This chapter describes the package `hvalidator` which provides all functions for validating XML documents represented as `XmlTree`. The validation process basically consists of three phases. First the DTD is validated, after this the document is validated. In the last step the document is transformed: missing default values are added and attribute values are normalized. Unlike other popular XML validation tools the validation functions return a list of errors instead of aborting after the first error was found.

While the modules from package `hparser` use a monadic approach, these modules are written purely functional. Validation is done by filter functions of type `XmlFilter`. These filters return a list of errors or an empty list if no errors are detected.

4.1. Module hierarchy

The only public module for validating XML documents is the module `Validation`. It exports all functions for validating and transforming XML documents. `Validation` is basically a helper module, which combines the functionalities of the internal modules taking part at validation and transformation.

Internal modules

`DTDValidation`

Provides functions for validating the DTD of XML documents.

`DocValidation`

Provides functions for validating the elements and attributes of XML documents.

`DocTransformation`

Provides functions for transforming XML documents after they have been validated. The transformation phase is part of the validation process.

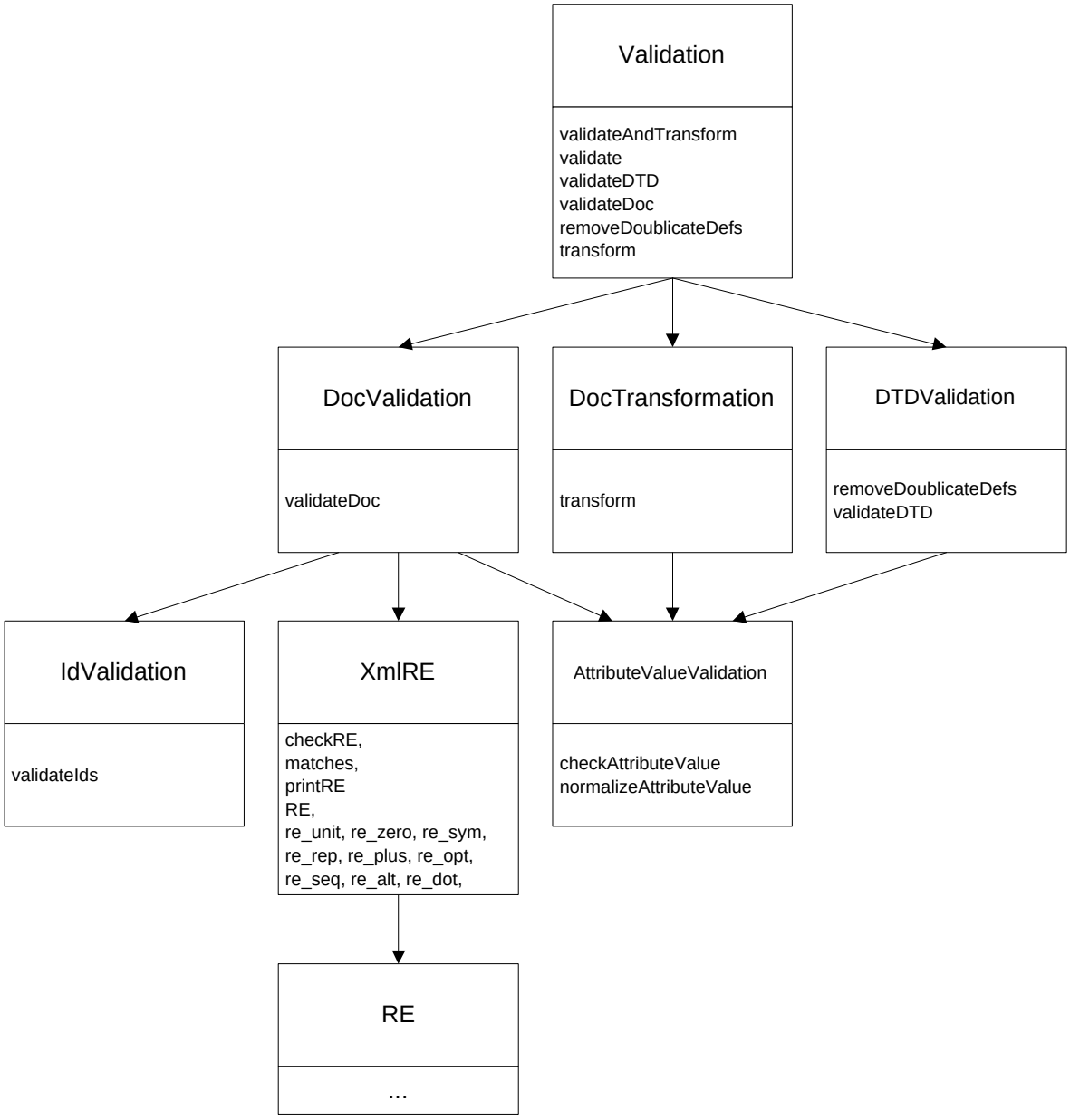
`AttributeValueValidation`

Provides functions to normalize attribute values and to check if the attribute value matches the lexical constraints of its type.

`XmlRE` and `RE`

Support `DocValidation` by validating the content model of an element. The algorithm used is based on derivatives of regular expressions.

Figure 4-1. Modules of package hvalidator



4.2. Creating a validating XML parser

The only public module for validating XML documents is the module `Validation`. It exports several functions for validating XML documents, parts of XML documents and transforming them.

`validateAndTransform :: XmlSFilter`

Combines validation and transformation of a document. If errors or fatal errors occurred during validation, a list of errors is returned. Otherwise the transformed document is returned.

`validate :: XmlSFilter`

Checks if the DTD and the document are valid.

`validateDTD :: XmlSFilter`

Checks if the DTD is valid.

`validateDoc :: XmlSFilter`

Checks if the document corresponds to the given DTD.

`transform :: XmlSFilter`

Transforms the document with respect to the given DTD. Validating parsers are expected to normalize attribute values and add default values. This function should be called after a successful validation.

The following example shows how the functions `validate` and `transform` can be used in an XML processing application. The document is valid, if `validate` returns an empty list or a list containing only errors of type *warning*. If the list contains errors of type *error* or *fatal error*, the document is not valid. If the document is valid the document is transformed and displayed to the user.

```
printMsg :: XmlTrees -> XmlTrees -> IO()
printMsg errors doc
    = if null ((isError +++ isFatalError) $$ errors)
        then do
             if null errors
               then
                  putStrLn "The document is valid."
                  putStrLn (xmlTreesToString $ transform doc)
             else do
                  putStrLn "The document is valid, but there were warnings:"
                  putStrLn (xmlTreesToString $ transform doc)
                  putStrLn (showXErrors errors)
        else do
             putStrLn "The document is not valid. List of errors:"
             putStrLn (showXErrors errors)



main :: IO()
main
    = do
      doc <- parseDoc "invalid.xml"
      printMsg (validate doc) doc
      return ()
```

Calling the module `ValidateExample` from the directory *example* of the Haskell XML Toolbox with the invalid document *invalid.xml* produces the following error messages.

Example 4-1. Validating a document with errors

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<!DOCTYPE a [
<!ATTLIST a  att1  CDATA  #IMPLIED>
<!ELEMENT a  (z, c?)>
<!ELEMENT b  EMPTY>
<!ELEMENT c  (#PCDATA)>
]>

<a att2="test">
    <y/>
    <c>hello world</c>
</a>

The document is not valid. List of errors:
Warning: The element type "z", used in content model of element "a", is not declared.
Error: The content of element "a" must match ( "z" , "c"? ). Element "z" expected,
       but Element "y" found.
Error: Attribute "att2" of element "a" is not declared in DTD.
Error: Element "y" not declared in DTD.
```

4.3. Validation of the Document Type Definition

Validation of the DTD is done by the module `DTDValidation`. Notations, unparsed entities, element declarations and attribute declarations are checked if they correspond to the constraints of the XML 1.0 specification [WWW01].

The following checks are performed:

DTD

- Error: There exists no DOCTYPE declaration in the document.

Notations

- Error: A notation was already specified.

Unparsed entities:

- Warning: Entity is declared multiple times. First declaration is used.
- Error: A referenced notations must be declared

Element declarations:

- Error: Element is declared more than once
- Error: Element was specified multiple times in a content model of type mixed content.
- Warning: Element used in a content model is not declared.

Attribute declarations:

- Warning: There exists no element declaration for the attribute declaration.

- Warning: Attribute is declared multiple times. First declaration will be used.
- Warning: Same Nmtoken should not occur more than once in enumerate attribute types.
- Error: Element already has an attribute of type ID. A second attribute of type ID is not permitted.
- Error: ID attribute must have a declared default of #IMPLIED or #REQUIRED.
- Error: Element already has an attribute of type NOTATION. A second attribute of type NOTATION is not permitted.
- Error: Attribute of type NOTATION must not be declared on an element declared EMPTY.
- Error: A notation must be declared when referenced in the notation type list for an attribute.
- Error: The declared default value must meet the lexical constraints of the declared attribute type.

Each check is done by a separate function, which takes the child list of the `XDTD DOCTYPE` node as input and returns a list of errors. Some functions can optionally take some further arguments to have access to context information, e.g. when validating unparsed entities, a list of all defined notations is needed. The result of validating the DTD is a concatenated list of the results of all validation functions.

The following example shows a filter function for checking the validity constraint: "No Notation on Empty Element" (section 3.3.1 in XML 1.0 specification [WWW01]). It means that an attribute of type NOTATION must not be declared for an element declared EMPTY. A notation attribute is a way to give an application a clue how the content of an element should be processed. The notation might refer to a program that can process the content, e.g. a base-64 encoded JPEG. Because empty elements cannot have contents, attributes of type notation are forbidden. The function `checkNoNotationForEmptyElement` is initialized with a list of all element names declared EMPTY. The constructed filter is then applied to all `XDTD ATTLIST` nodes of type NOTATION that have been selected from the DTD by another filter function.

Example 4-2. Validation that notations are not declared for EMPTY elements

```
checkNoNotationForEmptyElement :: [String] -> XmlFilter
checkNoNotationForEmptyElement emptyElems nd@(NTree (XDTD ATTLIST al) _)
    = if elemName ‘elem‘ emptyElems
      then err ("Attribute \""++ attName ++"\" of type NOTATION must not be "++
                "declared on the element \""++ elemName ++"\" declared EMPTY.") nd
      else []
      where
      elemName = getAttrValue1 a_name  al
      attName  = getAttrValue1 a_value al

checkNoNotationForEmptyElement _ nd
    = error ("checkNoNotationForEmptyElement: illegal parameter:\n" ++ show nd)
```

The validation functions cannot check if a content model is deterministic. This requirement is for compatibility with SGML, because some SGML tools can rely on unambiguous content models. XML processors may flag such content models as errors, but the Haskell XML Toolbox does not. It does not need deterministic content models for checking if the children of an element are valid (see Section 4.7).

4.4. Validation of the document subset

After validation of the DTD the document has to be validated. By validating the document, validation of the elements and their attributes is meant. This process is split into two phases: validation of the elements and their attributes and validation of attributes of types ID, IDREF and IDREFS.

Involved modules

`DocValidation`

Validation of the elements and their attributes. Validation of content models is delegated to the module `XmlRE`.

`IdValidation`

Validation of attributes of types ID, IDREF and IDREFS.

Before the document is validated, a lookup-table is built on the basis of the DTD. This table maps element names to their validation functions. The validation function for an element is a set of `XmlFilter` functions, which take the current element as input and return a list of errors. If the element meets the validation constraints, an empty list is returned. Each validation constraint, listed in the following, is represented by a single filter. All these filters are combined by the combinator `+++`, which concatenates the results of filters, each filter uses copy of state. This approach for validation is very flexible. It is very easy to add new functions without affecting the other ones.

After the initialization phase of building the validation filters and the lookup table, the whole document is traversed in preorder and every element is validated by its validation filter.

Validation of elements:

- Error: Element is not declared in DTD.
- Error: Root element must match value declared in DOCTYPE.
- Error: Children of element do not match its content model.

Validation of attributes:

- Error: Attribute is not declared.
- Error: Attribute is specified multiple times.
- Error: Attribute is of type #REQUIRED, but was not specified.
- Error: Attribute is of type #FIXED, but specified value differs from fixed value.
- Error: Value of attribute does not match the lexical constraints of its type.

The module `IdValidation` provides functions for checking special ID/IDREF/IDREFS constraints. First it is checked if all ID values are unique. All nodes with ID attributes are collected from the document, then it is validated that values of ID attributes do not occur more than once. During a second iteration over the document it is validated that all IDREF/IDREFS values match the value of some ID attribute. For both checks a lookup-table, which maps element names to their validation functions, is built like it is done in the module `DocValidation`.

Validation of ID/IDREF/IDREFS:

- Error: Value of type ID must be unique within the document.
- Error: An attribute of type IDREF/IDREFS references an identifier that does not exist in the document.

The following example shows a filter for validating that all attributes of an element are unique (Unique AttSpec, section 3.1 XML 1.0 specification [WWW01]). It is a filter, which takes an element - an `NTree XTag` node - checks if its attributes are unique, and returns a list of errors or an empty list.

Example 4-3. Validation that all attributes are unique

```
noDoublicateAttributes :: XmlFilter
noDoublicateAttributes n@(NTree (XTag name al) _)
    = doubles $ reverse al
      where
      doubles :: TagAttrl -> XmlTrees
      doubles ((attrName,_):xs)
          = if (lookup attrName xs) == Nothing
            then doubles xs
            else (err ("Attribute \""++ attrName ++"\" was already specified "++
                       "for element \"++ name ++"\".") n)
                 ++
                 doubles xs

      doubles [] = []

noDoublicateAttributes n
    = error ("noDoublicateAttributes: illegal parameter:\n" ++ show n)
```

4.5. Transformation of the document subset

Transformation of the document is done by the module `DocTransformation`. The transformation should be started after validating the document.

Tasks:

- Adding default values of attributes
- Normalizing attribute values
- Sorting all attributes in lexicographic order

Before the document can be transformed, a lookup-table is built on the basis of the DTD which maps element names to their transformation functions. The transformation functions are `XmlFilter` functions, which take a node and return the transformed node. After the initialization phase the

whole document is traversed in preorder and every element is transformed by its transformation filter from the lookup-table. The result is a new, transformed `XmlTree`.

4.6. Validation of attribute values

The module `AttributeValueValidation` provides all functions for validating attribute values. It provides a function for checking if the attribute value meets the lexical constraints of its type and a function for normalizing attribute values. These functions have been moved into an own module, because they are needed when the DTD subset and the document subset are validated and when the document is transformed. The function for checking the attribute value is again an `XmlFilter` function.

4.7. Derivatives of regular expressions

The following section describes how the algorithm for validating the element content works. It is implemented in the modules `XmlRE` and `RE`. The most commonly-used technique to solve this problem is based on finite automaton.

Another algorithm, based on derivatives of regular expressions, is much better suited for implementation in a functional language. This technique does not need to backtrack, like some NFA-based algorithms do and when combined with lazy evaluation and memorization it can be very efficient.

The algorithm of derivating regular expressions was first described by Janusz A. Brzozowski [Brzozowski64]. Joe English suggested using this algorithm for validating element contents in XML. He gave a short description of his ideas combined with a small Haskell program fragment [WWW26].

Mark Hopkins wrote some small C programs based on this algorithm. A description of the algorithm and working source code is available from the comp.compilers archive [WWW27]. James Clark describes an algorithm for RELAX NG validation [WWW28] that bases on derivatives, too.

4.7.1. Description

Validating the element content is an instance of regular expression matching problem. Basically the content model of an element describes a regular expression. If given the regular expression e and the content c it has to be checked if c is in L(e)? L(e) denotes the language accepted by the regular expression e.

Informal described, the algorithm works as follows. A regular expression is applied to a sentence. The remaining regular expression is checked if it can be derived to an empty sequence. If it can be derived to an empty sequence, the sentence matches the regular expression. Otherwise the sentence is not described by the regular expression.

The algorithm can be seen as working on an infinite DFA with states labeled by regular expressions instead of symbols as in the canonical construction. The final states are those states for which the regular expression can be derived to an empty sequence. The transition function is a function, which derives the regular expression by a single symbol.

4.7.2. Examples

The regular expression *foobar* should be derived with respect to the string "foo". After applying the regular expression to this string, the remaining regular expression is *bar*. This regular expression cannot be derived to an empty string, because it expects the characters ’b’, ’a’ and ’r’. This means that the string *bar* is not described by the regular expression. (Short form: foo\foobar = bar)

The regular expression *a** should be derived with respect to the string "aa". After applying the regular expression to the string, the regular expression *a** remains. This expression can be derived to an empty string, because the *-operator indicates that none or more ’a’ characters are expected. This means that the regular expression matches the string. (Short form: aa\a* = a*)

4.7.3. Realization in Haskell

The following code was adopted from Joe English [WWW26], but has been extended to support meaningful error messages, to accept any single symbol (dot operator) and to work with data types of `XmlTree`.

Regular expressions are defined by the algebraic data type RE.

```
data RE a =
    RE_ZERO String          --’ L(0)   = {} (empty set, or failure with message)
    | RE_UNIT               --’ L(1)   = { [] } (empty sequence, or success)
    | RE_SYM a              --’ L(x)   = { [x] }
    | RE_DOT                --’ accept any single symbol
    | RE_REP (RE a)         --’ L(e*)  = { [] } ‘union‘ L(e+)
    | RE_PLUS (RE a)        --’ L(e+)  = { x ++ y | x <- L(e), y <- L(e*) }
    | RE_OPT (RE a)         --’ L(e?)  = L(e) ‘union‘ { [] }
    | RE_SEQ (RE a) (RE a)  --’ L(e,f) = { x ++ y | x <- L(e), y <- L(f) }
    | RE_ALT (RE a) (RE a)  --’ L(e|f) = L(e) ‘union‘ L(f)
    deriving (Show, Eq)
```

The core of the algorithm is the `delta` ("derivative") function. It takes a regular expression `re` and a symbol `s`. The function returns a new expression that matches all sentences that are a suffix of sentences in L(`re`) beginning with `s`. The function works by simple case wise analysis.

```
delta :: (Eq a, Show a) => RE a -> a -> RE a
delta re s = case re of
    RE_ZERO m        -> re_zero m
    RE_UNIT          -> re_zero ("Symbol "++ show s ++" unexpected.")
    RE_SYM sym
        | s == sym   -> re_unit
        | otherwise  -> re_zero ("Symbol "++ show sym ++" expected, but "++
                                 "symbol "++ show s ++" found.")
```

```
    RE_REP e         -> re_seq (delta e s) (re_rep e)
    RE_PLUS e        -> re_seq (delta e s) (re_rep e)
    RE_OPT e         -> delta e s
    RE_SEQ e f
        | nullable e -> re_alt (re_seq (delta e s) f) (delta f s)
        | otherwise  -> re_seq (delta e s) f
    RE_ALT e f       -> re_alt (delta e s) (delta f s)
    RE_DOT           -> re_unit
```

The function `matches` derives a regular expression with respect to a sentence by using the higher-order function `foldl` which folds a function into a list of values.

```
matches :: (Eq a, Show a) => RE a -> [a] -> RE a
matches e = foldl delta e
```

The auxiliary function `nullable` tests if a regular expression matches the empty sequence. If regular expressions are compound ones and the outer expression cannot be itself derived to an empty sequence, the sub-expressions have to be tested if they are nullable.

```
nullable ::  (Show a) => RE a -> Bool
nullable (RE_ZERO _)  = False
nullable RE_UNIT      = True
nullable (RE_SYM _)   = False
nullable (RE_REP _)   = True
nullable (RE_PLUS e)  = nullable e
nullable (RE_OPT _)   = True
nullable (RE_SEQ e f) = nullable e && nullable f
nullable (RE_ALT e f) = nullable e || nullable f
nullable RE_DOT       = True
```

The function `checkRE` checks if an input matched a regular expression. The regular expression `RE_ZERO` indicates that an error occured during derivation and that the regular expression does not match the given sentence.

```
checkRE :: (Show a) => RE a -> String
checkRE (RE_ZERO m) = m
checkRE re
    | nullable re = ""
    | otherwise   = "Input must match " ++ printRE re
```

The constructor functions `re_seq`, `re_opt` and so on, used in the `delta` function, are not shown. Without these constructor functions the intermediate regular expressions would grow very rapidly. The constructors simplify them at each step, e.g. by replacing a sequence of epsilons (`RE_UNIT`) by a single epsilon.

The described `delta` function is not the one used for validating `XmlTree` types. There exists a special function for validating `XmlTree` nodes in the module `XmlRE`.

4.7.4. Conclusions

The algorithm of derivating regular expressions can be expressed in Haskell very naturally: regular expressions are represented by a special data type, sentences are just a list of symbols. Pattern-matching and Haskell's standard functions for list processing allow a very clear and short implementation.

The algorithm can handle ambiguous regular expressions very well. If a regular expression consists of sub-expressions, all sub-expressions are derived by a certain symbol. If for example the ambiguous regular expression `(a , b) | (a , c)` is derived with respect to the sentence "`ab`", the algorithm will construct the regular expression `(b | c)` after deriving the original expression by the symbol `a`. After deriving the remaining regular expression by the symbol `b`, the regular expression `(RE_UNIT | RE_ZERO)` remains. This expression can be derived to the empty sequence and therefore the regular expression matches the sentence.

Haskell's lazy evaluation avoids the evaluation of the whole regular expression. The expression has only to be evaluated as much that `nullable` can calculate an answer.

Chapter 5. Conclusion

5.1. XML conformance of the parser

The validating XML parser of the Haskell XML Toolbox was elaborately tested with the XML Test Suites (XML TS) from the W3C [WWW02].

These XML TS consist of over 2000 test files for the XML 1.0 specification [WWW01] and an associated test report. The test report contains background information on conformance testing for XML as well as test descriptions for each of the test files.

The test suite consists of two basic test types: Binary Tests and Output Tests. Binary conformance tests are documents that should be accepted or rejected by the parser. Output tests are tests for valid documents which are paired with a reference file as the canonical representation [WWW03] of the input file. By comparing the output of the XML parser with the reference file, it can be ensured that the parser provides the correct information.

There exist four different tests cases:

Valid Documents

: The XML parser is required to accept these documents. For most test cases a reference file exists.

Invalid documents

: A validating XML parser has to report a violation of some validity constraint. The reported error must meet the test case description.

Not well-formed documents

: An XML parser has to report a fatal error. The reported error must meet the test case description.

Optional errors

: An XML parser is permitted to ignore these errors, or to report them. If an error is reported, the error message must meet the test case description.

By using these tests, it could be ensured that the validating XML parser of the Haskell XML Toolbox covers most aspects of the XML 1.0 specification [WWW01]. Because the XML TS is not complete, it is not possible to claim conformance with the specification itself, but only with the XML TS. At the moment the parser passes about 95% of these test cases, see Section A.2 for further details.

5.2. The Haskell XML Toolbox in comparison to HaXml and HXML

In this section the Haskell XML Toolbox is compared with HaXml [WWW21] and HXML [WWW25]. Many valuable ideas of HaXml and HXML have been adopted by the Haskell XML Toolbox. It is not the intention of this section to run down these great projects. The intention is to show how their ideas have been extended and generalized.

The Haskell XML Toolbox, HaXml and HXML differ in the way how XML documents are represented in Haskell. First the approaches of HXML and HaXml are introduced, after this the data model of the Haskell XML Toolbox is compared with them.

In HXML, XML document subsets are represented as a `Tree` of `XMLNodes`. The hierarchical structure of XML documents is modeled by the generic tree data type `Tree`. This type does not distinguish between inner nodes and leafs. Leafs are just nodes with an empty list of children.

Example 5-1. Document subset in HXML

```
data Tree a  = Tree a [Tree a]

type XML  = Tree XMLNode
data XMLNode =
      RTNode                -- root node
    | ELNode GI AttList     -- element node: GI, attributes
    | TXNode String         -- text node
    | PINode Name String    -- processing instruction (target,value)
    | CXNode String         -- comment node
    | ENNode Name           -- general entity reference
      deriving Show
```

DTDs are modeled totally different in HXML. They are represented by named fields and are not stored in the tree model where the document subset is stored.

Example 5-2. DTD subset in HXML

```
data DTD = DTD {
    elements :: FM.FM Name ELEMTYPE,    -- elemtps / element types
    attlists :: FM.FM Name [ATTDEF],    -- elemtype.attdefs
    genents  :: FM.FM Name EntityText,  -- general entities
    parments :: FM.FM Name EntityText,  -- parameter entities
    notations:: [DCN],                  -- nots/notations
    dtdname  :: Name                    -- name (document type name)
}  deriving Show
```

HaXml's representation of XML documents differs totally from the approach HXML and the Haskell XML Toolbox use. Instead of modeling XML documents with a generic tree type, HaXml uses a more data centric approach. The whole structure of XML documents is modeled by different algebraic data types. There exist special types for almost each production of the XML 1.0 specification [WWW01]. XML documents are modeled by the data type `Document`, which consists of a `Prolog` and the document subset, an `Element`. This data model distinguishes clearly between leafs and inner nodes. Leafs are types which constructors do not take any arguments.

Example 5-3. XML documents in HaXml

```
data Document = Document Prolog (SymTab EntityDef) Element
data Prolog   = Prolog (Maybe XMLDecl) (Maybe DocTypeDecl)
data XMLDecl  = XMLDecl VersionInfo (Maybe EncodingDecl) (Maybe SDDecl)
...
```

The document subset is modeled in HaXml by the algebraic types `Element` and `Content`. An `Element` has an attribute list and a list of `Content` types. If the content list is empty, the element is a leaf. Together these types define a mutually recursive, multi-branch tree.

Example 5-4. Document subset in HaXml

```
data Element   = Elem Name [Attribute] [Content]

type Attribute = (Name, AttValue)
data Content   = CElem Element
               | CString Bool CharData
               | CRef Reference
               | CMisc Misc
```

HaXml introduced the idea of using filter functions and combinators for processing parts of the XML data model. The examples from the previous chapters show that this approach is very powerful and flexible. The whole XML parser of the Haskell XML Toolbox bases on filters. The filters of HaXml work for nodes of the type `Content`.

Example 5-5. The filter type of HaXml

```
type CFilter   = Content -> [Content]
```

The Haskell XML Toolbox uses the most generic data model in contrast to HaXml and HXML. Its data model is a generalization of the data models discusses above.

The generic tree data model `NTree` of the Haskell XML Toolbox forms the basis for representing XML documents in Haskell. This type does not distinguish between inner nodes and leafs. Leafs are just nodes with an empty child list. The most important aspect is that this generic tree data model represents a whole XML document, including the DTD subset, the document subset and all other logical units of XML. Two algebraic data types `XNode` and `DTDElem` are used to represent all logical units of XML.

Example 5-6. XML documents represented in the Haskell XML Toolbox

```
data NTree  node = NTree node (NTrees node)
type NTrees node = [NTree node]

type XmlTree  = NTree  XNode
type XmlTrees = NTrees XNode

data XNode =
      XTag TagName TagAttrl
    | XDTD DTDElem TagAttrl
    | ...
```

```
data DTDElem =
      DOCTYPE
    | ELEMENT
    | ATTLIST
    | ...
```

HXML uses the same generic data model as the Haskell XML Toolbox for representing the document subset, but DTDs are represented by a totally different model: named fields. HaXml uses special types for XML's logical units. This leads to the fact that the DTD subset is modeled totally different than the document subset.

The advantage of representing the whole XML document by one generic data type lies in the fact that one unique design for processing the whole document can be used. Because all logical parts of XML are modeled by one generic data model, filters (see Section 2.3) can be used to process the whole XML document and not only parts as in HaXml and HXML.

HaXml's filters can only work on the type `Content` that just represents a small part of XML documents, the document subset. If one wants to process other parts of an XML document, one cannot use filters any more, but has to implement special functions. The same applies for HXML.

The generalization used in the Haskell XML Toolbox makes the design of applications that process whole XML documents very uniform. In effect the design of the whole XML parser of the Haskell XML Toolbox bases on filters. Merging of the internal DTD part and external DTD part is done by filters, checking the validity constraints of DTDs and document subsets is done by filters, or processings like transforming the whole `XmlTree` back to XML is done by filters.

5.3. Conclusions and future work

The developed XML parser shows that the functional approach accomplishes the task of parsing and validating XML by using fewer lines of code and producing a very short and compact program in contrast to imperative languages. The packages `hdom`, `hparser` and `hvalidator` contain only about 9.000 lines of code including lots of HDoc [WWW30] comments.

Although the program is compact, the code is understandable and maintainable, because the code is more succinct and it follows a clear and simple design. The Haskell XML Toolbox introduced the very general tree data type `XmlTree` for representing whole XML documents in Haskell. This general data model makes it possible to base all processings of XML documents on filters. The whole XML parser, presented in this thesis, bases on this uniform design.

Writing a validating XML parser is a quite complex task. It must cope with different encodings, correct processing of entities and of course validation. Functional programming helps master this complexity better than other methods.

In Haskell functions are just values and have no side effects. These qualities allow an easy use of higher-order functions that take functions as arguments, return functions as a result or do both. The filter combinators, which have been adopted from HaXml, form a powerful library for combining

filter functions. Because all filters of the Haskell XML Toolbox share the same type, it is possible to combine them freely with the use of filter combinators. All details of manipulating the `XmlTree` data structure are hidden in these higher-order functions. In effect these filter combinators define problem specific control structures that make it possible to program on a very high abstraction level. Errors can be reduced, because programmers can use the filter combinators as standard functions for processing the `XmlTree`.

Because functions are just values in Haskell, they can be constructed at runtime. The XML parser introduced in this thesis makes an extensive use of creating parameterized filter functions during runtime. The whole validation process bases on this design.

It can be quite useful to use functional programming paradigms when writing programs in imperative languages. Functional paradigms like higher-order functions for abstractions can be done in Java by defining an interface that has only one function. This interface can be passed to other functions or returned by a function. In C function pointers can be used for this task. But imperative languages are not designed to support functional programming styles, so they cannot actively support its paradigms. The main focus of these languages lies on the fact *how* a problem is solved, e.g. the order in which computations are performed.

Unfortunately functional programming and functional programming languages are not very popular. Two standard examples for functional programming are using a spreadsheet program like Excel and querying a database with SQL. Another field where functional programming dominates is transformation of SGML and XML documents. The Document Style Semantics and Specification Language (DSSSL) [WWW09] bases on the functional programming language Scheme and is very popular in the world of SGML publishing. The Extensible Stylesheet Language (XSL) [WWW07] is its corresponding part for XML publishing. Functional programming languages are very well equipped for this task, because the transformation process is a functional mapping from a structural document as input to a formatted representation as output.

Sometimes it is said that functional programming languages lack of libraries. We do not know any validating XML Parser written in Haskell and hope that the framework of the Haskell XML Toolbox will be a useful tool for XML processing applications written in Haskell. The parser supports almost fully the XML 1.0 specification with the exception of namespaces.

The Haskell XML Toolbox introduces a powerful approach for processing XML in Haskell. It generalizes the ideas of HaXml and HXML. Whole XML documents are represented as a tree of different nodes. This tree can be processed in a uniform way by using filter functions and filter combinators.

Lots of great ideas of the projects HaXml and HXML have been taken into this project. We want to thank their members for their great work and emphasis that all three projects are enrichments for the Haskell community.

The Haskell XML Toolbox project will be maintained and enlarged at the University of Applied Sciences Wedel. One student already wrote an XSLT processor on the basis of this project. Another student is writing a program using the Haskell XML Toolbox for deriving Java classes from DTDs.

Appendix A. User handbook

A.1. System requirements

The Haskell XML Toolbox was developed with *Hugs Version December 2001* [WWW20] and the *Glasgow Haskell Compiler (GHC) 5.04* [WWW19] on Linux systems. Testing on Windows systems has not been done yet.

If you want to use the Haskell XML Toolbox in your own applications, just add the modules from the directories `hdom`, `hparser`, `hvalidator` and `parsec` to the path of your compiler or interpreter. It is planned to provide special GHC packages of the Haskell XML Toolbox in the near future.

To compile binary versions of the programs included in the Haskell XML Toolbox, the *Glasgow Haskell Compiler* and *GNU make* are needed.

To build the API documentation you must have installed *HDoc* [WWW30]. If you run `make hdoc` in the root directory of the Haskell XML Toolbox, the documentation will be generated and placed in the directory `doc/hdoc`.

A.2. Missing features and known problems

Unfortunately the XML parser of the Haskell XML Toolbox does not support fully the XML 1.0 specification at the moment. We are working hard to implement the missing features in the near future.

Missing features:

- Namespaces are not supported.
- Protocols like "http" and "file" are not yet supported due to a lack of libraries. We think of using the `curl` command instead.
- General external entities are not yet processed.
- General entities in default values of attributes are not yet substituted.

Known problems:

- Line numbers are not yet reported for validity constraint errors.
- Under Hugs 98 only: The parser suffers a serious space fault when parsing large documents.

A.3. Directory structure

`doc`

Contains the documentation: this thesis and the generated HDoc documentation

`examples`

Contains some example applications for using the parser, filter functions and filter combinators. Includes `HXmlParser`, a command line well-formedness checker and validator.

`hdom`

Sources of package `hdom`.

`hparser`

Sources of package `hparser`.

`hvalidator`

Sources of package `hvalidator`.

`parsec`

Sources of the free monadic parser combinator library Parsec. Parsec is already distributed with Hugs and GHC, but we had some problems with different behaviors of these libraries.

A.4. HXmlParser - Well-formedness checker and validator

The Haskell XML Toolbox comes with a command line tool for checking if XML documents are well-formed and valid. To build this tool with GHC, run `make HXmlParser` in the root directory of the Haskell XML Toolbox. This will produce the binary file `HXmlParser` that will be stored in the root directory. The sources of this tool are located in the directory `examples`.

When checking an invalid XML file, `HXmlParser` will report errors to *stderr* and quit with an exit code of *"-1"*. If there are no errors in the document, the tool will just quit without any message.

If you run `HXmlParser` without any arguments, the usage information is displayed:

```
Usage:
  --source "filename"  - source file (required)
  --encoding "charset" - document encoding
  --trace "level"      - trace level (0-4)

  -h                   - this help
  -v                   - display a message if document is well-formed or valid
  -w                   - well-formed check only

  -op                  - output parsed document
  -opt                 - output XmlTree of parsed document
  -oph                 - output Haskell type of parsed document
```

```
Only for validity checks:
  -ot                  - output transformed document
  -ott                 - output XmlTree of transformed document
  -oth                 - output Haskell type of transformed document

  -oc                  - output canonicalized document
  -oct                 - output XmlTree of canonicalized document
  -och                 - output Haskell type of canonicalized document
```

A.5. Check the XML parser with the XML Test Suites

The XML parser of the Haskell XML Toolbox was elaborately tested with the XML Test Suites from the W3C [WWW02]. If you want to run these checks on your own, you can use the tool `RunTestCases` that is distributed with the Haskell XML Toolbox.

First you have to download the test cases from the W3C [WWW02] and extract them in an own directory. To test the XML parser with these test cases, you have to build the program `RunTestCases`. Executing `make RunTestCases` in the root directory of the Haskell XML Toolbox will compile this program.

The XML Test Suites contain test cases from different organizations and people. `RunTestCases` must be called with a test case description file as argument from the directories: `ibm`, `oasis`, `sun` or `xmltest`.

Perhaps a dummy root node must be added to these files to form a valid XML document. `RunTestCases` cannot process the root file that combines all theses description files.

When being executed, `RunTestCases` will ask what type of test shall be performed. The type entered must correspond to the tests listed in the test case description file. `RunTestCases` will only execute these test cases.

If the option "show detailed information" is set, `RunTestCases` will display the `XmlTree` in addition if a test fails.

Example A-1. Executing RunTestCases

```
RunTestCases ibm_oasis_valid.xml

Which kind of test shall be performed?
([1] - valid | 2 - invalid | 3 - not-wf | 4 - error):

Show detailed information? (j/[n])
```

A.6. Performance and profi ling

Time and space profiling is very useful to optimize programs and remove bottlenecks. We already added the ability of profiling to the XML parser, but did not use this information yet. The main focus of the parser lays on a clear and easy design at the moment.

A.6.1. Usage information

The validating XML parser can be analyzed with the time and space profiling system of the Glasgow Haskell Compiler (GHC). Cost centers have been added for all main computations, so that it could be measured which parts of the parser consume most time and memory.

The tool `Profiling` can be used to run these tests with the XML parser. It can be built by executing `make Profiling` in the root directory of the Haskell XML Toolbox.

`Profiling` must be executed with an XML file as test input and several different parameters that influence GHC's profiling behavior. These parameters are described in depth in the documentation of GHC.

Four predefined profiling types have been added to the `Makefile` in the root directory of the Haskell XML Toolbox. The input file of `Profiling` can be set by the variable `PROFILING_INPUT`.

Predefined profiling targets:

`make runprofiling1`

Time and allocation profiling - Generate profiling information in XML format and display them with the tool *ghcprof* of GHC.

`make runprofiling2`

Time and allocation profiling - Generate profiling information in text format and display them with *less*.

`make runprofiling4`

Heap profiling - Break down the graph by the cost-center stack which produced the data.

`make runprofiling4`

Heap profiling - Break down the graph by the retainer set.

A.6.2. How to interpret the results

This section will show how to interpret the results from the profiling tests. The W3C Recommendation of the Extensible Hypertext Markup Language (XHTML) and its Strict DTD was chosen as test input. The test machine was a Pentium 4 with 1.6 GHz and 256 MB memory, running Debian 3.0.

The results from time and space profiling show that most time and memory is consumed by the function `procesDTD`. This function provides substitution of parameter entities in the internal and external DTD part. The next resource intensive function is `parseXmlDoc` that does the parsing of an XML document and constructs an `XmlTree` representation from it.

```
total time  =        1.26 secs   (63 ticks @ 20 ms)
total alloc = 102,664,004 bytes  (excludes profiling overheads)


COST CENTER                    MODULE               %time %alloc

processDTD                     HdomParser            27.0   45.6
parseXmlDoc                    HdomParser            19.0   29.3
buildAllValFcts                DocValidation         14.3    5.0
validateAttributes             DTDValidation         12.7    0.5
processGeneralEntities         HdomParser             7.9    9.6
MAIN                           MAIN                   6.3    5.0
buildAllTransFcts              DocTransformation      4.8    1.8
IdVal.validateIds              Validation             4.8    1.1
validateElements               DTDValidation          3.2    0.2
getXmlContents                 HdomParser             0.0    1.1
```

The ranking of the cost centers might have changed dramatically if the profiling tests are performed with other documents. In this case, parameter entities are exhaustively used in the Strict DTD of XHTML. It is not astonishing that the function that handles their substitution takes the most time and space. Another important aspect is the ratio of the DTD subset and the document subset. If for example the DTD subset is very large, its processing functions might take a large percentage of time and space consumption, too.

Heap profiling produces graphs that show the memory consumption of the cost centers during execution time. These graphs show very clearly how the program execution behavior of Haskell programs differs from programs written in imperative languages. Haskell uses lazy evaluation when evaluating expressions. Arguments to functions are evaluated only when they are needed. Functions only calculate their result as much as it is needed by other functions that called them.

Lazy evaluation leads to the fact that Haskell programs do not have a strict sequential evaluation order, but that they resemble a stream. The graphs show very clearly this execution behavior. First the parsing functions are started, after producing enough output, processing of the DTD starts. In the end the validation process starts calculations. The graphs show that almost all functions work in parallel after they have enough input.

Appendix B. MIT License

The Haskell XML Toolbox shall be used be many people and is therefore distributed under the MIT License.

Bibliography

Print resources

[Thompson99] *The craft of Functional Programming: 2nd Edition*, Simon Thompson, Addison Wesley, 1999, 0-201-34275-8.

[Hudak00] *The Haskell School of Expression: Learning functional programming through multimedia*, Paul Hudak, Cambridge University Press, 2000, 0-521-64408-9.

[Brzozowski64] *Derivatives of Regular Expressions*, Janusz A. Brzozowski, Journal of the ACM, Volume 11, Issue 4, 1964.

Online resources

[WWW01] *Extensible Markup Language (XML) 1.0*, W3C, 2000, http://www.w3.org/TR/2000/REC-xml-20001006 .

[WWW02] *Extensible Markup Language (XML) Conformance Test Suites*, http://www.w3.org/XML/Test/ .

[WWW03] *Canonical XML, Version 1.0*, John Boyer, http://www.w3.org/TR/2001/REC-xml-c14n-20010315 .

[WWW04] *XML Syntax Quick Reference*, Mulberry Technologies, Inc., http://www.mulberrytech.com/quickref/XMLquickref.pdf .

[WWW05] *Document Object Model (DOM)*, W3C DOM Working Group, http://www.w3.org/DOM/ .

[WWW06] *JDOM*, http://www.jdom.org .

[WWW07] *The Extensible Stylesheet Language (XSL)*, XSL Working Group, http://www.w3.org/Style/XSL/ .

[WWW08] *XML Schema*, C. M. Sperberg-McQueen, Henry Thompson, http://www.w3.org/XML/Schema .

[WWW09] *ISO/IEC 10179:1996 - Document Style Semantics and Specification Language*, James Clark, http://www.jclark.com/dsssl/ .

[WWW10] *Simple API for XML (SAX)*, David Brownell, http://www.saxproject.org/ .

[WWW11] *Haskell.org*, http://www.haskell.org .

[WWW12] *Haskell Brooks Curry*, J. O'Connor, E. F. Robertson, http://www-gap.dcs.st-and.ac.uk/~history/Mathematicians/Curry.html .

[WWW13] *About Haskell*, Based on a paper by Simon Peyton Jones., December 2001, http://www.haskell.org/aboutHaskell.html .

[WWW14] *A Gentle Introduction to Haskell, Version 98*, Paul Hudak, John Peterson, Joseph Fasel, June 2000, http://www.haskell.org/tutorial/index.html .

[WWW15] *The Haskell 98 Report*, Simon Peyton Jones, John Hughes et al., http://www.haskell.org/onlinereport/ .

[WWW16] *The Haskell 98 Library Report*, Simon Peyton Jones, John Hughes et al., http://www.haskell.org/onlinelibrary/ .

[WWW17] *The Haskell Bookshelf* , http://www.haskell.org/bookshelf/ .

[WWW18] *Why Functional Programming Matters*, John Hughes, http://www.math.chalmers.se/~rjmh/Papers/whyfp.html , 1984.

[WWW19] *GHC*, http://www.haskell.org/ghc .

[WWW20] *Hugs*, http://www.haskell.org/hugs .

[WWW21] *HaXml: Haskell and XML*, Malcolm Wallace, Colin Runciman, http://www.cs.york.ac.uk/fp/HaXml/ .

[WWW22] *Haskell and XML: Generic Combinators or Type-Based Translation?*, Malcolm Wallace, Colin Runciman, John Boyer, http://www.cs.york.ac.uk/fp/HaXml/icfp99.html .

[WWW23] *The HaXml functional programming model for XML*, David Mertz, October 2001, http://www-106.ibm.com/developerworks/xml/library/x-matters14.html .

[WWW24] *Functional Programming and XML*, Bijan Parsia, 2000, http://www.xml.com/lpt/a/2001/02/14/functional.html .

[WWW25] *HXML*, Joe English, http://www.flightlab.com/~joe/hxml/ .

[WWW26] *How to validate XML*, Joe English, 1999, http://www.flightlab.com/~joe/sgml/validate.html .

[WWW27] *Regular Expression Package. Posted to comp.compilers.*, Mark Hopkins, 1994, http://compilers.iecc.com/comparch/article/94-02-109 ftp://iecc.com/pub/file/regex.tar.gz .

[WWW28] *An algorithm for RELAX NG validation*, James Clark, February 2002, http://www.thaiopensource.com/relaxng/derivative.html .

[WWW29] *Parsec*, Daan Leijen, http://www.cs.uu.nl/~daan/parsec.html .

[WWW30] *HDoc*, Armin Groesslinger, http://www.fmi.uni-passau.de/~groessli/hdoc/ .

Affidavit

I hereby declare that this master thesis has been written only by the undersigned and without any assistance from third parties.

Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Hamburg, 2002-09-02

```
---------------------------------------------------------------
Place, Date Signature
```

.