A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science (MSc.)

A Cookbook for the Haskell XML Toolbox with Examples for Processing RDF Documents

Manuel Ohlendorf

January 6, 2007

Computer Science Department

Contents

List of Figures

Listings

1. Preface

The processing of the Extensible Markup Language (XML) has become a typical task for programs, since XML is a standard language for exchanging data between applications. All languages used in the World Wide Web to describe data are based on XML. These are XHTML, a language to make data human-readable, XSLT, a style-sheet language for XML to transform XML documents in any other format, or RelaxNG, a schema language for XML to define the structure of a document more fine grained then it can be done with Document Type Definitions (DTD).

The functional programming language Haskell is one of the most popular one since then it has been defined as a standard in 1998 [Jones et al. 1998] and has been further developed to a powerful language until now. More and more professional applications are based on Haskell and there is a large set of additional libraries providing all kind of special functionalities. Different XML parsers belong to this set of libraries and one of them is the Haskell XML Toolbox. The project was initiated by Prof. Dr. Uwe Schmidt from the University of Applied Science Wedel and it was firstly presented by Martin Schmidt's Master Thesis [Schmidt 2002]. The Haskell XML Toolbox consists of a XML parser, a module to validate XML documents, a module to use XPath expressions and a XSLT module, which is not finished yet. The concept of the Haskell XML Toolbox has been fundamentally changed recently, in this way, that the functions for manipulating the XML document, the high-level programming interface, are now based on arrows.

The aim of this thesis is to show and describe this new approach and compare it with the former one. An RDF/XML parser is written as an example application using the Haskell XML Toolbox, to show the usage of the new concept. This parser is furthermore extended by the possibility to search the parse result with a query language.

Chapter two describes the structure and the concept of the Haskell XML Toolbox. Simple examples are used to describe the application of the various functions. Chapter three firstly gives an introduction to RDF and then shows how to process RDF/XML documents. Furthermore a simple query language to search in the RDF is introduced and implemented. The last chapter compares the Haskell XML Toolbox with other XML parsers written in Haskell and concludes the advantages of the new concept.
The reader of this document should be familiar with the functional programming language Haskell.

2. HXT – Haskell XML Toolbox

2.1. Introduction

The Haskell XML Toolbox is a very modern and elegant validating Extensible Markup Language (XML) 1.0 (Second Edition) [XML] parser. Since the first release it has been extended by several additional modules. These are a XPath module and a XSLT parser module which is not finished yet. One of the main changes which were recently made was the implementation of a new arrow interface. The first versions of the Haskell XML Toolbox were using monads to provide I/O and state handling. The processing functions were based on the idea of *filters*. Every function was of the same type and could be therefore combined with several special operators. John Hughes showed in the paper "Generalizing monads to arrows" [Hughes 2000] that it is in several situations more elegant to use arrows instead of monads and that arrows give the possibility to define a special notion of computation. What arrows are, what kind of advantages they bring to the Toolbox and how they are used is described later in the thesis.
Before starting with more complex examples, first of all the data structure of the Toolbox has to be explained.

2.2. Basic Data Types

2.2.1. NTree

The most common way to represent the hierarchical structure of XML documents is to use trees. Trees furthermore can be modelled with lists. The data type `NTree`, defined in the module `Data.Tree.NTree.TypeDefs`, is the main data structure used throughout the whole parser. It is a generic data type and an instance of the type class `Tree` defined in `Data.Tree.Class`, where the structure of a tree is specified.
`NTree` is a n-ary ordered tree also called rose tree because it can be widely ramified. The trees are defined as a node with a list of child nodes. Leafs of the tree are nodes which do not have any children. The type synonym `NTrees` is a shortcut for node lists. The following listing shows the defition of `NTree` and `NTrees`:

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

type NTrees a = [NTree a]
```

As `NTree` is generic, it cannot be used only for XML documents, but to represent any kind of tree. The following examples use `NTree` with `Int` in order to explain the usage of this data type. Listing 2.1 shows an example of a tree with integers as child-nodes.

Listing 2.1: NTree with Int

```
intTree :: NTree Int
intTree = NTree 1 [ NTree 2 [ NTree 5 [],NTree 6 []
                            ],
                    NTree 3 [],
                    NTree 4 []
                  ]
```

This is a simple tree where the nodes are of type Int. For visualisation, the function `formatTree` can be used. The result of this function is as follows:

```
---1
   |
   +---2
   |   |
   |   +---5
   |   |
   |   +---6
   |
   +---3
   |
   +---4
```

2.2.2. XmlTree

The type-specific version of the generic `NTree` is `XmlTree` defined in the module `Text.XML.HXT.DOM.TypeDefs`. Together with `XmlTrees` it defines a general recursive data type for XML documents:

```
type XmlTree = NTree XNode

type XmlTrees = NTrees XNode
```

Every data which is stored in `XmlTree` has to be of type `XNode`. This data type is used to represent every possible logical unit of a XML document. This can be for example a simple element, a comment or a text node.
The Haskell XML Toolbox is a validating parser and can therefore also handle Document Type Definitions (DTD). DTDs define the structure of XML documents. This definition is compared with the document while parsing. Only those documents are valid which fit into the definition of the DTD. To process these definitions `XNode` can also be a DTD definition.
Before the data type `XNode` is introduced, two types which are used by `XNode` have to be described. These are `Attributes` and `QName`.

```
type Attributes = AssocList String String
```

`AssocList` is simply a key value association list, implemented as an unordered list of pairs for storing all kind of properties and features of the DTD parts.
In order to support namespaces [XML-NS] in XML documents, the Haskell XML Toolbox uses the data type `QName`. Namespaces give the possibility to avoid element collisions, which are elements with the same name but different meanings. This can happen when two different applications process the same XML document. Both applications then use or expect an element with the same name but understand it differently.

Namespaces are defined with a specific *Uniform Resource Identifier* (URI) and a prefix for this namespace once in a XML document. The elements of this namespace then use this prefix in their element name. This name is also called *qualified name*.
`QName` is divided into three sections: the prefix, the local part which is the name of the element or attribute and the URI of the namespace.

```
data QName = QN {
  namePrefix   :: String
  localPart    :: String
  namespaceUri :: String
}
```

Finally, the algebraic data type `XNode` defines, along with the described types `Attributes` and `QName`, the basic nodes and leaves for all kinds of XML's logical units. Moreover, the data type `DTDElem` defines the constructors for the DTD declarations.

Instead of explaining every single constructor of `XNode`, only the most important ones should be described here. The constructor `XTag` defines an element which can be an inner node, if the element has children or a leaf, if the element is empty. `QName`, as explained earlier, is the name of the XML tag and the type `XmlTrees` is the list of attributes of the element. Because attributes are also stored with the data type `XmlTree`, every function for processing XML documents can be used also for processing the attributes. In this case `XNode` has the constructor `XAttr`.
`XError` is an internal extension and not a XML component. It stores the level and message of errors which may occur during parsing.

```
data XNode
= XText String
| XCharRef Int
| XEntityRef String
| XCmt String
| XCdata String
| XPi QName XmlTrees
| XTag QName XmlTrees
| XDTD DTDElem Attributes
| XAttr QName
| XError Int String
deriving (Eq, Ord, Show, Read, Typeable)
```

The following example shows how the simple XML document in listing 2.2 looks like in the described data structure of the Haskell XML Toolbox.

Listing 2.2: Simple XML Document

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?pi this is a processing instruction?>
<test attr="hello">
        world!
        <test2/>
</test>
```

The graph shown in the next example is the XML document after parsing. The root node '/', which has been generated, has several new attributes added by the parser. They contain information about the document and the command line parameters of the parser. Furthermore, the parser has generated a DTD for this document with the general predefined entities *lt, gt, amp, apos* and *quote*.

Listing 2.3: Graph of the XML Document

```
---XTag "/"
   |   "trace"="4"
   |   "source"="test.xml"
   |   "status"="0"
   |   "parse-html"="0"
   |   "validate"="1"
   |   "issue-errors"="1"
   |   "issue-warnings"="1"
   |   "check-namespaces"="0"
   |   "canonicalize"="1"
   |   "preserve-comment"="0"
   |   "remove-whitespace"="0"
   |   "module"="getXmlContents"
   |   "transfer-Protocol"="file"
   |   "transfer-URI"="file://example.xml"
   |   "transfer-Status"="200"
   |   "transfer-Message"="OK"
   |   "version"="1.0"
   |   "encoding"="ISO-8859-1"
   |   "transfer-Encoding"="ISO-8859-1"
   |
   +---XPi "xml"
   |   |   "version"="1.0"
   |   |   "encoding"="ISO-8859-1"
   |
   +---XDTD DOCTYPE []
   |   |
   |   +---XDTD ENTITY [("name","lt")]
   |   |   |
   |   |   +---XCharRef 38
   |   |   |
   |   |   +---XText "#60;"
   .   .
   .   . (Here are the definitions of the other predefined entities)
   .   .
   |
   +---XText "\n"
   |
   +---XPi "pi"
   |   |   "value"="this is a processing instruction"
   |
   +---XText " \n"
   |
   +---XTag "test"
```

```
|   |   "attr"="hello"
|   |
|   +---XText "\nworld!\n"
|   |
|   +---XTag "test2"
|   |
|   +---XText "\n"
|
+---XText "\n"
```

The Haskell XML Toolbox also provides a function to print out the internal representation, i.e. the Haskell code of the tree, which is sometimes very helpful while debugging a program.

2.2.3. Class XmlNode – Basic Interface to NTree and XNode

The class `XmlNode` in the module `Text.XML.HXT.Arrow.XmlNode` defines all the functions for processing `XNode` and `NTree`. Since it is a type class, it only contains some default implementations but the real functionality is defined in the instances `XNode` and `NTree` of this class.
The processing functions of `XmlNode` can be divided into four different categories: *predicates*, *selectors*, *modifiers* and *constructors*.
Predicates are functions of type *bool*. They are used to test specific properties of elements. A simple example is the function `isText` which checks if a node is a text node. The following listing shows the type definition and the implementations of `XNode` and `NTree`.

```
isText :: a -> Bool

-- the XNode implementation
isText (XText _)  = True
isText _          = False

-- the NTree implementation
isText = isText ∘ getNode
```

`XmlNode` provides various selector functions, in order to access parts of the tree. These functions use the `Maybe`-type to facilitate that nothing can be returned in case that a node processed by a selector function does not have the right expected properties. Again, a simple example is `getText` which either returns the text value of a text node, or nothing if it is not a text node.

```
getText :: a -> Maybe String

-- XNode implementation
getText (XText t) = Just t
getText _         = Nothing

-- NTree implementation
getText   = getText ∘ getNode
```

This example shows that the `NTree`-implementation of `XmlNode` is very simple. It just combines the function `getNode` with the `getText`-implementation of the type returned by `getNode`.
The third category is made up of the modifier functions that alter nodes and their attributes. There are always two versions of these functions: one which takes a function as parameter to change the specific node and the other which just takes a value to set the new value of a node.
The constructor functions allow to create new element nodes. The function `mkText` for example takes a character string and returns a text node.
But for implementing a program which should process a XML document somehow, the functions which were just described are not very helpful. Things like I/O or state and failure handling are not provided by these functions but necessary for real programs. Therefore, the Haskell XML Toolbox delievers a set of arrow classes which implements these functionalities like I/O and state handling. The class `XmlNode` is merely the interface for the arrow classes to the data types of the Haskell XML Toolbox. These arrow classes will be described in the following sections.

2.3. Arrows in Haskell

2.3.1. Introduction

The processing functions provided by the Haskell XML Toolbox are based on arrows. Like monads, arrows allow to define different notions of computation, but in a much more general manner. With arrows one can define computation with some kind of static state handling or computation that consumes multiple inputs. There are different subclasses of arrows, which not only give the possibility of choice and feedback but also a special arrow which is equivalent to monads.
The arrow classes provide combinators which allow a point-free programming style but sometimes it can be awkward for programming specific instances. Therefore Ross Paterson has introduced a point-wise notation in his paper [Paterson 2001] for arrows which

is supported by the Haskell compiler GHC [GHC]. The following sections introduce the arrow libraries of Haskell and show how to use the arrow notation.

2.3.2. Main Arrow Class

The `Arrow` class consists of two functions `arr` and `(>>>)`. The purpose of the function `arr` is to convert a simple function into an arrow function; the operator `(>>>)` provides composition for arrows. This is analogous to the usual `Monad` class – it has a way of creating a monad function out of a pure computation with `return` as well as a way of sequencing computation with `(>>=)`.
Besides these two functions, the `Arrow` class provides a third combinator `(&&&)` which does not have an equivalent function in the `Monad` class. Actually, the functionality provided by `(&&&)` is already in the composition function of monads. This is to make the second arrow of `(&&&)` dependant on the *effects* of the first arrow which is not possible with `arr` and `(>>>)`. The next listing shows the `Arrow` class with its three functions:

```
class Arrow arr where
  arr   :: (a -> b) -> arr a b
  (>>>) :: arr a b -> arr b c -> arr a c
  (&&&) :: arr a b -> arr a c -> arr a (b,c)
```

The operator `(&&&)` takes two arrows and returns another arrow with the results of the two input arrows paired as output. Thus this operator allows to sequence two computations. A simple example is to apply two functions delivering integers simultaneously to the input and sum their results up. With the `(&&&)` operator it is very easy to define an arrow doing this:

```
addA f g = f &&& g >>> arr (uncurry (+))
```

To make the implementation as easy as possible the `Arrow` class actually contains more then those three functions. The operator `(&&&)` is defined with much simpler functions and only one of them has to be implemented to get the full functionality. This function is called `first` which lifts an arrow to operate on pairs by feeding just the first components through the given arrow and leaving the second one untouched. The other functions, which do not have to be implemented, shall not be described here. The type definition of `first` is:

```
first :: arr a b -> arr (a,c) (b,c)
```

A simple example to show the functionality of arrows are *stream functions* taken from the paper of John Hughes [Hughes 2004]. In order to make a data type an instance of the `Arrow` class, it has to be a *newtype* rather than a type synonym. The type definition of stream functions (data type `SF`) is as follows:

Listing 2.4: Stream Function Definition

```
newtype SF a b = SF {runSF :: [a] -> [b]}
```

The instance definition for the `Arrow` class is not very complicated. Stream functions are functions representing computiation from list to list. A simple function converted with `arr` to a stream function just needs to be called by the higher-order function `map`. This converts the simple function to a function from a list to a list.
The implementation of the composition of stream functions uses the composition for simple functions, which is predefined in the `Prelude` of Haskell.

Listing 2.5: SF Arrow Instance

```
instance Arrow SF where
  arr f = SF (map f)
  SF f >>> SF g = SF (f >>> g)
  first (SF f) = SF (unzip >>> first f >>> uncurry zip)
```

The definition of `first` also uses `first` for simple functions.
Stream functions have to be invoked via `runSF`. The following example shows the execution of a stream function and the result of it in the interpreted version of Haskell:

Listing 2.6: Sample Call of SF

```
Stream> runSF (arr (+1)) [1..5]
[2,3,4,5,6]
```

This example illustrates that the function (+1) is applied to every element of the list. Another simple example and very useful operation for stream functions is to delay the stream by one element, adding a new element at the beginning of the stream:

Listing 2.7: Delay Function

```
delay :: a -> SF a a
delay x = SF (x:)
```

These examples above show some of the compelling advantages of arrows. A monadic program always takes its input via the parameters of a function and therefore only in one way. It is not possible to change this by varying the monad. By using arrow programs instead, it depends on the particular arrow how the program takes its input. This is because arrow computations are parameterised over their output as well as their input type. The stream functions, as a simple example, take a stream of values rather than a single value which cannot be represented as a monad.

2.3.3. Additional Arrow Classes

There are several additional arrow classes which add special features to the main arrow class. Those additional functionalities are not included in the main arrow class, because not every arrow has all these properties. The four classes shortly introduced here are `ArrowChoice`, that provides an operator to make an arrow conditional on the output of another, `ArrowZero` and `ArrowPlus`, which provide operations for failure and failure handling and `ArrowApply`, that actually makes arrows as powerful as monads.

ArrowChoice

The operator (|||) provides conditionals for arrows. It uses the `Either` type as the input of the choice operator so that the `Left` and `Right` values can carry different types of data. The class is defined as:

```
class Arrow arr ⇒ ArrowChoice arr where
  (|||) :: arr a c -> arr b c -> arr (Either a b) c
```

Like the operator `(&&&)` on pairs the choice operator is defined with simpler operators and only one of them has to be implemented when creating an instance of the `ArrowChoice` class. This operator is called `left`. A call of `left f` passes inputs tagged `Left` to `f`, passes inputs tagged `Right` straight through, and tags output from `f` with `Left`. The type definition of `Left` is:

```
left :: arr a b -> arr (Either a c) (Either b c)
```

ArrowZero

The class `ArrowZero` has only one arrow which represents the case of a failure. What counts as a failure is defined in the implementation of this arrow. As will be described

later, this can be, for example, the empty list for computations over lists. The definition of the class is:

```
class Arrow a ⇒ ArrowZero a where
  zeroArrow :: a b c
```

ArrowPlus

The class `ArrowPlus` provides the combinator `(<+>)` in order to handle this failure described by the class `ArrowZero`. Its definition is shown in the next listing:

```
class ArrowZero a ⇒ ArrowPlus a where
  (<+>) :: a b c -> a b c -> a b c
```

`(<+>)` takes the two arrows, applies the input to both of them simultaneously, and fuses the output of the arrows into one. Therefore, this combinator can also be seen as the logical *Or* like the combinator `(>>>)` represents the logical *And*.

ArrowApply

The idea of the class `ArrowApply` is to provide higher-order programming with arrows. This makes it possible to construct arrows which receive other arrows in their input and invoke them. Hence, the class has a new arrow `app` which is analogous to the "apply" function:

```
class Arrow arr ⇒ ArrowApply where
  app :: arr (arr a b, a) b
```

The instance definitions for pure functions are fairly simple:

```
instance ArrowApply (->) where
  app (f,x) = f x
```

The `ArrowApply` class is equivalent to the `Monad` class. In order to do the same with arrows that can be done with monads, a special implementation of the class `Monad` is needed. The type which does this is a computation of `a` as an arrow from the empty tuple to `a`:

```
newtype ArrowMonad arr a = ArrowMonad (arr () a)
```

With this type `return` and `(>>=)` can be defined as follows:

```
instance ArrowApply a ⇒ Monad (ArrowMonand a) where
  return x = ArrowMonad (arr (const x))
  ArrowMonad m >>= f =
    ArrowMonad (m >>>
                arr (λx -> let ArrowMonad h = f x in (h, ()))
                >>> app)
```

The function `f` which returns an arrow is turned into an arrow outputting an arrow (`h`) with `arr` and then `app` invokes the result.
Hughes says that arrows supporting `app` are of relatively little interest because arrow types which correspond to a monad can be much easier replaced by a monad directly. Only those types that cannot be represented as a monad are "interesting" arrow types. Still `app` is a useful arrow which is also used in the Haskell XML Toolbox to solve some problems with the point-free programming style.

2.3.4. Arrow Syntax

Although point-free programming is very elegant and readable, it is sometimes clearer to give names to the values being manipulated. In monadic programs this pointed programming is well-supported by the do-notation. To provide the same do-notation for arrow programming Ross Paterson has designed a language extension [Paterson 2001] which is introduced in this section. This language extension is implemented by using a preprocessor. It translates the arrow notation into standard Haskell code before it is compiled.
The new syntax adds a new form of expression called the *arrow abstraction* of the form `proc pat -> cmd` where `proc` is a new binding operator; the body of such an expression is called a *command*. The arrow application is the simplest form of command: `a <- expr` where `expr` is a Haskell expression to be the input to the arrow `a`. The translation of this into the point-free style is the following:

```
proc pat -> a -< expr  =  arr (λpat -> expr) >>> a
```

The `do` notation for arrows looks quite the same as the notation for monads. For the arrow notation `do` blocks are nothing else then commands. The statement `x <- e` in a `do` block means that it binds the name `x` to the output of the command `e`. As an example, the `addA` arrow defined above can be rewritten:

```
addA :: Arrow a ⇒ a b Int -> a b Int -> a b Int
addA f g = proc z -> do
    x <- f -< z
    y <- g -< z
    returnA -< x + y
```

The input `z` is fed to the two arrows `f` and `g`, and their outputs are bind to `x` and `y`. Finally, the result of `x + y` is sent to the arrow `returnA`. The `-<` as the arrow application operator means that it is the tail feather of an arrow. The binding `x <- f -< e` looks as though `e` is being fed through an arrow labeled with `f` to be bound to `x`.
The arrow `returnA` is called *entity arrow*. This arrow is analogous to `return` in the monad notation and is defined:

```
returnA :: Arrow a ⇒ a b b
returnA = arr id
```

As explained and shown in the example, the arrow notation can be very helpful for programming with arrows. Especially if the output of an arrow is needed several times or is combined with other outputs of arrows. Nevertheless, during the next sections the arrow syntax is not used when the arrow modules of the Haskell XML Toolbox are explained, because the code in the point-free style is much easier to understand. Furthermore the Haskell XML Toolbox provides special combinators that allow to use an argument in two places. But, of course, every example can also be expressed with this new syntax.

2.4. Main Arrow Modules

Now that the idea of representing computation with arrows and the data structure of the Haskell XML Toolbox have been explained, the next sections show the different arrows used by the parser. As already described, the best way to model the tree structure of XML is to use lists. Furthermore, the intention of all the parts of the Haskell XML Toolbox is to be as generic as possible. So, like the arrow library of Haskell where the arrows are separated in different classes providing special functionalities, the Haskell XML Toolbox contains several arrows – all for a special purpose. One of these arrow classes provide computation over lists. This is the `ArrowList`, the most important arrow class. The class `ArrowIf` adds operators for conditional cases to the computation over lists and the class `ArrowTree` adds arrows for processing trees implementing the earlier introduced `Data.Tree.class` interface. Finally, the class `ArrowXml` combines all classes and provides all kind of arrows for processing XML.

2.4.1. ArrowList – List Processing

Before the new arrow classes has been included in the Haskell XML Toolbox, the processing functions were all of the same type and were called *filter*. Every filter was of the type `XmlTree -> [XmlTrees]` and the idea of it was to make them easy to combine and to be able to handle several ways of output. With a list as output type, the failure of a predicate was represented by an empty list and it was also possible to handle more than one result as the output.
The idea of the class `ArrowList` is basically the same, but it is much more flexible. In what way will be described in chapter 4.
A list arrow is a function, which has a list of results as output for a given input. If the list only contains one element, the arrow represents a normal function. In the case of an empty list, the function is undefined for the given argument. Or, if the function is a predicate, it represents the boolean value **False**; for none empty list a **True**. The nondeterministic case is covered by a list with more than one element as result. So, the list arrows represent computation as relations instead of partial functions.
An implementation of the class `ArrowList` is used in order to provide several examples. This is the data type `LA` defined in the module `Control.Arrow.ListArrow` and it is a function from a single argument `a` to a list of `b`:

```
newtype LA a b = LA { runLA :: a -> [b] }
```

This already shows the considerable advantage of list arrows over the filter approach. While every filter has to have the same type of result, namely a list of `XmlTrees`, the output type of list arrows is generic. The list returned by the arrow can contain any kind of element. It was often a problem with the filters that one needs different return types then `XmlTrees`, which was impossible. To solve this problem, everything was encapsulated by `XmlTrees`. Thus, type errors were not detected by the type checker because everything was of type `XmlTree`. Now, with arrows the type system of Haskell can be used to prevent these errors because the return type is not fixed anymore. Hence, the arrow approach provides the same flexibility as simple functions, while it is easier to use than filters and prevents a lot of errors.
Two very important arrows which are actually aliases for existing arrows are `this` as the identity arrow and `none` as the zero arrow. They are defined as follows:

```
this    :: a b b
this    = returnA

none    :: a b c
none    = zeroArrow
```

The arrow `this` is an alias for `returnA` described in section 2.3.4 and `none` for `zeroArrow` from the class `ArrowZero`. It is not necessary to use those arrows instead of the original one, but the code becomes much more readable and logical since they are used for combining arrows with the combinators described earlier. For the arrow `none` an implementation of `ArrowZero` is needed. This is done by the list arrow data type as the following:

```
instance ArrowZero LA where
  zeroArrow   = LA (const [])
```

To complete the definition of the data type `LA` as an arrow, it also needs the implementation of the class `Arrow`. The constructor arr creates a list arrow from a normal function by returning the result of it in a list. The combinator `(>>>)` is defined by combining the two functions with function composition and applying `concatMap` to it. `concatMap` creates a list-from-a-list-generating function by application of this function on all elements in a list passed as the second argument. The function `first` which is used to define `(&&&)` returns an arrow from a pair to a list of pairs where the function `f` is applied only on the first element of the input pair:

```
instance Arrow LA where
  arr f          = LA ( λ x -> [f x] )
  LA f >>> LA g = LA ( concatMap g ∘ f )
  first (LA f)   = LA ( λ ~(x1, x2) -> [ (y1, x2) | y1 <- f x1 ] )
```

These are only those functions which do not have a default implementation. For efficiency, the other functions and operators like `second` and `(&&&)` are also implemented but shall not be described here. Now that the data type `LA` is defined as an arrow, the arrows of the class `ArrowList` and their implementation can be introduced.
The arrows, which need an implementation are `arrL`, `arr2A`, `isA` and `(>>.)`. `arrL` is, like arr, a constructor. It builds a list arrow from a function which has a list as the output type:

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

Another constructor is `arr2A` which creates a two-argument arrow from a single-argument one:

```
arr2A :: (b -> a c d) -> a (b, c) d
```

Besides the identity and the zero arrow, `isA` is an important but simple arrow. It builds an arrow from a predicate function which returns a single list containing the input if the predicate holds and the empty list if not:

```
isA  :: (b -> Bool) -> a b b
```

The last function which needs to be implemented is the combinator `(>>.)`. It converts the result of a list arrow with a given function into another list:

```
(>>.) :: a b c -> ([c] -> [d]) -> a b d
```

The class `ArrowList` has to be implemented first, before the next examples can use the data type `LA`. The following listing shows this instance definion:

```
instance ArrowList LA where
  arrL        = LA
  arr2A f     = LA ( λ ~(x, y) -> runLA (f x) y )
  isA p       = LA ( λ x -> if p x then [x] else [] )
  LA f >>. g = LA ( g ∘ f )
```

The function `isTwo` is a predicate which returns **True** if the input was two or it returns **False** if not. To build a list arrow out of it, the arrow `isA` takes this function as input.

Listing 2.8: Example Predicate

```
isTwo :: Int -> Bool
isTwo n = n == 2

testIsTwo a = runLA ( isA isTwo ) a
```

Running this example in the interpreter, it would return the empty list if `a` is not two and a single list containing two if the input is two.
Another example is the list arrow `addSomething` which takes an extra argument of type `int` and is shown in listing 2.9.

Listing 2.9: Arrow with Extra Parameter

```
addSomething :: (ArrowList a) ⇒ Int -> a Int Int
addSomething x = arr (+x)

testAdd = runLA (addSomething 2) 2
```

The call of `testAdd` would generate the integer four in a single list as output. Using `addSomething` as input for the constructor `arr2A` would generate an arrow of type (`a (Int, Int) Int`). With this function, testAdd could be defined as the following, which also generates a single list with four as output:

Listing 2.10: arr2A Example

```
testAdd = runLA (arr2A addSomething) (2,2)
```

Sometimes it is practical to create constant arrows. This can be achieved by the the constructor `constA`. It does the same as `const` for simple functions which generates a constant function out of the input. The definition of `constA` is:

```
constA :: c -> a b c
constA = arr ∘ const
```

The following example shows the use of the combinator `(>>.)`. First, the combinator `(<+>)` and the constructor `constA` generate a list with two elements. Second the list is reversed by the function reverse and `(>>.)`:

Listing 2.11: (>>.) Example

```
ListArrow> runLA (const 2 <+> const 4 >>. reverse) []
[4,2]
```

The constructor arrow `arrL` is equivalent to the data type constructor `LA`. It builds a list arrow from a function with a list as result. The arrow `arr2L` does the same, but it generates a list arrow with two arguments:

```
arr2L :: (b -> c -> [d]) -> a (b, c) d
arr2L = arrL ∘ uncurry
```

Besides the constructor `arr`, the class `ArrowList` also provides constructors to generate arrows from functions with more than one argument. These constructors are `arr2`, `arr3` and `arr4` where the numbers in their names specify the number of arguments the functions can have:

```
arr2 :: (b1 -> b2 -> c) -> a (b1, b2) c
```

They are very helpful in combination with the pair operator `(&&&)` which generates an arrow with a pair as result. Taking two arrows `a1` and `a2`, sequencing them and then

combining the result with the binary function `f` would look like this:
`a1 &&& a2 >>> arr2 f`
Sometimes it is necessary to convert a nondeterministic into a deterministic arrow when the list of results must be manipulated. This conversion can be achieved by the combinator `listA`:

```
listA :: a b c -> a b [c]
listA af = af >>. (:[])
```

An example for `listA` is the function `collectAndSort` which takes an arrow and combines the deterministic version of it with the function `sort`:

Listing 2.12: Determenistic Arrow

```
collectAndSort :: (ArrowList a, Ord c) ⇒ a b c -> a b c
collectAndSort collect = listA collect >>> arrL sort
```

Furthermore, the class `ArrowList` provides two generalisations for the arrow combinator `(<+>)` and `(>>>)` to ensure that the code remains readable and compact. Instead of long chains of arrows combined with those combinators, only a list of arrows is needed. These generalisations are `catA` and `seqA`:

```
catA :: [a b c] -> a b c
catA = foldl (<+>) none

seqA :: [a b b] -> a b b
seqA = foldl (>>>) this
```

An example of `catA` is to build a list of numbers by combining `constA` arrows and sorting them with the `collectAndSort` arrow:

Listing 2.13: Generalisation of (>>>)

```
runLA (collectAndSort (catA [constA 3, constA 1, constA 5])) []
  == [1,3,5]
```

Pointed Programming with ArrowList

The following groups of arrows all solve a problem of the point-free programming style: using an argument in two places is not possible. One solution is to use the new arrow

notation described above or the following arrows where the combinator `applyA` is the most important one. It uses the argument in two places. First, to compute an arrow with it and, second, to apply this new arrow to the input which is done by the `app` arrow of the `ArrowApply` class. This means that every data type which should implement the class `ArrowList` also has to implement `ArrowApply`. Therefore, the example data type `LA` for simple list processing also implements it. The definition of `applyA` is:

```
applyA :: a b (a b c) -> a b c
applyA f = (f &&& this) >>> app
```

The implementation of the class `ArrowApply` shows the following listing.

```
instance ArrowApply LA where
    app = LA ( λ (LA f, x) -> f x )
```

The combinator `applyA` is used to define several other arrows which all deal with the same problem of the point-free programming using values more than once. One of them is the infix operator `($<)`. The following listing shows its definition:

```
($<)     :: (c -> a b d) -> a b c -> a b d
g $< f     = applyA (f >>> arr g)
```

It computes the arrow `f` to get the parameter for the arrow with an extra parameter `g` from the input and applies the arrow `g` for all parameter values to the input which is very useful for joining arrows. So if `f` computes n values, the whole arrow computes n values but only if `g` is deterministic. If `f` fails, the whole arrow will fail.
There is also a binary version of `($<)`, a version taking three, and a version which takes four extra parameters. These are the operators `($<<)`, `($<<<)` and `($<<<<)`. Each of them solves the problem with the point-free programming. The next listing shows an example with simple list arrows over strings and the use of the binary operator `($<<)`:

Listing 2.14: Point-Wise Example

```
infixString :: String -> String -> a String String
infixString s1 s2 = arr (λ s -> s1 ++ s ++ s2)

runLA ( infixString $<< constA "y" &&& constA "z" ) "x"
```

The result of this list arrow would be the string *"yxz"*.
An arrow which offers a slightly different but significant functionality is the operator

`($<$)`. In contrast to `($<)` this operator applies all results of the second arrow sequentially to the input by the arrow with an extra parameter. This allows programming in a point-wise style in the second arrow, which again becomes necessary, when a value is needed more than once.
If it is essential to transform a single value step by step, this combinator is very useful. The second arrow collects the data for all steps and the arrow with an extra argument transforms the input step by step. The definition of the operator is the following:

```
($<$) :: (c -> a b b) -> a b c -> a b b
g $<$ f = applyA (listA (f >>> arr g) >>> arr seqA)
```

If `g` is a deterministic arrow (i.e. computes exactly one result) the results of `g $<$ f` and `g $< f` are equal. But if `g` computes more than one result the whole arrow only has one result because `f` is applied sequentially to the input for every result of `g`. Again, the arrow `addSomething` is used to show the functionality of the combinator:

```
runLA ( addSomething $<$ constA 2 <+> constA 3 ) 1
```

This arrow would compute the result `[5]` while the same arrow with the combinator `($<)` would compute the result `[3,4]`.
This is the introduction to the `ArrowList` class so far. Now, the other arrow classes provided by the Haskell XML Toolbox are introduced. Naturally, the arrows of `ArrowList` will be described more detailed later in more complex examples.

2.4.2. ArrowIf – Conditional Arrows

The class `ArrowIf` provides conditional combinators for list arrows and is defined in the module `Control.Arrow.ArrowIf`. All conditional operations are based on the idea that the result of an arrow is a list and the empty list represents **False** while a none-empty list represents **True**. For that reason, every data type implementing `ArrowIf` also has to implement `ArrowList`.
Two arrows `ifA` and `orElse` do not have a default implementation. The combinator `ifA` is the standard **if** lifted to list arrows. The first argument is the predicate arrow, the second the then-case and the third arrow describes the else-case. The type signature of `ifA` is the following:

```
ifA :: a b c -> a b d -> a b d -> a b d
```

The directional choice is provided by the combinator `orElse`. If the first arrow succeeds, then the result of it is returned, else the second arrow is applied to the input. In most

cases, the arrow `orElse` is used in the infix notation. Its type definition is:

```
orElse :: a b c -> a b c -> a b c
```

Only these two arrows need an instance definition in order to gain the entire functionality from `ArrowIf`. The data type `LA` which was already used for simple examples is also an instance of this class. The following listing shows the implementation:

Listing 2.15: LA Implementation of ArrowIf

```
instance ArrowIf LA where
  ifA (LA p) t e
    = LA ( λx -> runLA (if null (p x) then e else t) x )
  (LA f) `orElse` (LA g)
    = LA ( λx -> let res = f x in
                 if null res then g x else res )
```

Another useful arrow is `neg` which is the same as `not` for simple predicate functions and is used to negate arrows. The arrows `when` and `guards` are also combinators to make the application of arrows dependent on predicates. The call `f `when` g` means that, if the predicate `g` holds, the arrow `f` is applied, and if not, the identity filter `this` is returned. In contrast, `g `guards` f` means that when `g` does not hold, nothing is returned, and when `g` holds, `f` is applied.
The arrow `containing` tests whether the results of the first arrow are holding the predicate, in which case only those results are returned.
Case expressions in Haskell are very useful for handling multi-way branches of conditionals. This can also be done with list arrows using the arrow `choiceA`, which is a generalisation of `orElse`. The arrow `choiceA` uses an auxiliary data type `IfThen` with an infix constructor `(:->)` to deal with multi-way branches. The next listing displays an example and the definition of it. In the example, `p1` and `p2` are predicates and `exp1` to `exp3` are the expressions which should be applied:

Listing 2.16: choiceA Example

```
choiceA :: [IfThen (a b c) (a b d)] -> a b d

--example
choiceA [ p1   :-> exp1
        , p2   :-> exp2
        , this :-> exp3]
```

2.4.3. ArrowState

Dealing with states is a very important feature of programming with Haskell. These states can be some kind of information which is needed throughout the whole program, for example a counter for generating unique identifiers. In monadic programs, this state handling can be done by the class `MonadState`. For programs based on arrows, the Haskell XML Toolbox provides the class `ArrowState` to manage an explicit state. State arrows work similarly to state monads by threading a state value through the application of arrows. The definition of the class and its functions are shown in the next listing:

```
class Arrow a ⇒ ArrowState s a | a -> s where
    changeState :: (s -> b -> s) -> a b b
    accessState :: (s -> b -> c) -> a b c
    getState    :: a b s
    setState    :: a s s
    nextState   :: (s -> s) -> a b s
```

The arrows `changeState` and `accessState` take functions which change the state based on the old state and which access the state with a function using the arrow input as data for selecting state components, respectively. To read the complete state while ignoring the arrow input, the class provides the arrow `getState`. The arrow `setState` allows to overwrite the old state. Especially for consecutive states like identifiers, the arrow `nextState` is provided. It changes the state via a simple function and returns the new state value, while the arrow input is ignored.
To show examples of the state arrows, the data type `LA` is not appropriate anymore. A data type which implements the class `ArrowState` requires also special implementations of all other arrows like the `(>>>)` or `(&&&)` operators. This is because the state components always have to be taken into consideration throughout all arrow computations. Thus, the Haskell XML Toolbox provides the data type `SLA` defined in the module `Control.Arrow.StateListArrow`, which is a list arrow combined with state handling. The implementation of all arrow classes like `ArrowList` should not be described here, because it does not differ much from the implementation of LA. The only difference is that every arrow has to loop the state values through the application. The definition of the simple data type `SLA` is the following, where `s` represents the state:

```
newtype SLA s a b = SLA { runSLA :: s -> a -> (s, [b]) }
```

A simple example of generating consecutive numbers with the arrow `nextState` is presented by the following listing. The arrow `newId` takes an `Int` as input, increments this by one and returns the result as `String`.

```
newId :: SLA Int b String
newId = nextState (+1)
        >>>
        arr (('_':):) ∘ show)
test = runSLA 0 (newId <+> newId <+> newId) undefined
```

Invoking `test` in the interpreter would generate the output `(3,["_:1","_:2","_:3"])`, in which the first element of the tuple is the final state and the second is the list of result generated by the arrow `newId`. The function `undefined` is used as input for the arrow because the input is ignored by the state arrow.

2.4.4. ArrowIO

The support of I/O operations is, like state handling, indispensable for larger programs. The class `ArrowIO` provides arrows for lifting I/O actions to arrows. Furthermore, the module in which the class is defined, contains also a class called `ArrowIOIf` which allows to convert an I/O predicate to an arrow. The class `ArrowIO` only has some constructors to create different arrows. The one that has no default implementation is `arrIO` and is defined as:

```
arrIO :: (b -> IO c) -> a b c
```

There are several possible implementations of `ArrowIO`. One is the data type `IOLA` defined in the module `IOListArrow`, which combines computations over lists with I/O handling, and another data type is `IOSLA` which also provides state handling. The data type and instance definition of `IOLA` for the class `ArrowIO` is the following:

```
newtype IOLA a b = IOLA { runIOLA :: a -> IO [b] }

instance ArrowIO IOLA where
    arrIO cmd = IOLA ( λx -> do
                        res <- cmd x
                        return [res])
```

In addition to the constructor `arrIO`, there are several arrows which differ in the number of arguments. These are from `arrIO0`, which constructs an arrow from an I/O action without any parameter to the constructor `arrIO4`, taking an I/O action with four parameters.

2.4.5. ArrowTree – Tree Processing

The last class `ArrowTree` which adds special features to the Haskell XML Toolbox is essential for the XML parser. It provides arrows for processing trees which implement the `Data.Class.Tree` interface. This is the generic data type `NTree` and therefor also the type `XmlTree`. All functions of `ArrowTree` have default implementations and use list arrows for processing. That is why the simple data types like `LA` are also instances of `ArrowTree` without any extra implementation. The main arrows of `ArrowTree` are functions defined in `Data.Class.Tree` and lifted to arrows. Therefore, the definition of how `ArrowTree` actually processes trees depends on the implementation of `Data.Class.Tree`. These arrows are `getChildren`, `setChildren`, `changeChildren`, `getNode`, `setNode` and `changeNode`. The first three arrows process the child-nodes of the root of a tree by selecting the children, substituting them or editing the children with a given function, respectively. In order to process the attribute of the root of a tree, the second three arrows are provided and, again, allow to select, substitute or change the attribute.

The data type `NTree` as an instance of `Data.Class.Tree` provides all these functions and therefore also the type `XmlTree`, which means that these arrows also allow to process a whole document tree of a XML document. To show the functionality in some examples, the list arrow data type `LA` and the tree `intTree` defined in listing 2.2.1 is used. The function `addOne` increments every element in a list of `NTree` and its children by one. Handing this function over to `changeChildren`, will apply it to every child element of the root tree, as shown in listing 2.17.

Listing 2.17: changeChildren Example

```
addOne :: [NTree Int] -> [NTree Int]
addOne [] = []
addOne ((NTree i ys):xs)
  = (NTree (i+1) (addOne ys)) : addOne xs

testAddOne = runLA (changeChildren (addOne)) intTree
```

The result of `testAddOne` would be a tree in which, besides the root, every element is incremented by one. This can be achieved much easier because `ArrowTree` provides several compound arrows for traversing the whole tree with different strategies.

The arrow `processChildren` applies an arrow element-wise to all children of the root of a tree, collects the results of it and than substitutes the children with this result. With `processChildren`, the function `addOne` is much easier to define. Instead of a simple function, it is now defined as a list arrow:

Listing 2.18: processChildren Example

```
addOneA :: LA (NTree Int) (NTree Int)
addOneA = changeNode (+1) >>> processChildren addOneA

testAddOneA = runLA (processChildren addOneA) intTree
```

The result of `testAddOneA` would be, again, a tree in which every element has been incremented by one. This, however, can be done even easier than within the last example. The arrows `processTopDown` and `processBottomUp` recursively transform a whole tree by applying an arrow to all subtrees. This is done with a top down depth first traversal strategy by `processTopDown` and bottom up, depth first, leaves first and the root as last tree by `processBottomUp`. Therefore, it is not necessary to implement the recursion on one's own. Furthermore, there are the arrow `processBottomUpWhenNot` that stops the transformation if a predicate does not hold and the arrow `processTopDownUntil`, which stops the recursion if a tree is successfully transformed.
Now `addOneA` can be rewritten with one of the traversal arrows. Both of them can be used, because traversing the tree from top to bottom gives in this case the same result than from bottom to top:

Listing 2.19: processBottomUp Example

```
addOneA :: LA (NTree Int) (NTree Int)
addOneA = processBottomUp (changeNode (+1))

testAddOneA = runLA (processChildren addOneA) intTree
```

Several other arrows provide additional features for processing a tree. The arrow `replaceChildren` is similar to `processChildren`, but the children are replaced by new ones which are computed by processing the whole input tree. With `insertChildrenAt` and `insertChildrenAfter`, it is possible to determine in which place of the tree the new children should be inserted. The arrow `insertChildrenAt` takes an index where the computed list of trees should be inserted and `insertChildrenAfter` searches the insertion place with a predicate.
In order to search a whole tree for subtrees recursively, the class `ArrowTree` provides the arrows `deep` as a top down search, `deepest` as a bottom up search and `multi` also as a top down search. While `deep` and `deepest` stop when a tree is found which matches the predicate and returns it, `multi` moves on (when a matching tree is found) and returns all the subtrees for which the predicate holds.

2.4.6. ArrowXml

The classes `ArrowList`, `ArrowIf`, `ArrowState`, `ArrowIO` and `ArrowTree` together provide all the functionalities, needed by a powerful XML parser. The features described in the last sections are combined in the class `ArrowXml` defined in the module `Text.XML.HXT.Arrow.XmlArrow`. It provides a huge set of arrows to process XML documents. The class `ArrowDTD` is based on this interface, which contains special arrows for DTD processing. All the functions which are defined in the class `XmlNode` and described in section 2.2.3 are lifted to arrows, so that now the processing functions for XML documents are able to deal with I/O actions, global states and errors. Furthermore, the arrows of `ArrowXml` are also grouped in *predicates*, *selectors*, *constructors* and *modifiers*. Since instances of `ArrowXml` are also instances of `ArrowList`, all predicates return `this`, if the predicate holds, while returning `none` in case it fails (see section 2.4.2). In order to make the difference between the functions in `XmlNode` and `ArrowXml` clear, the next listings show how `isText` and `getText`, which were already introduced in section 2.2.3, are defined.

```
isText  :: a XmlTree XmlTree
isText  = isA XN.isText
```

Since `isText` is of type bool, it is handed over to the arrow `isA`, which generates a list arrow out of a predicate function. The qualified import name `XN` is used to differ between the function defined in the class `XmlNode` and the new local definition. The selector function `getText` returns the text of a text node and is lifted to arrows as follows:

```
getText :: a XmlTree String
getText = arrL (maybeToList  ∘ XN.getText)
```

In this listing, `arrL`, which creates an arrow from a list function, is used to lift `getText`. The function `getText` is transformed from the `Maybe` type to a list function with `maybeToList`, defined in the `Prelude` of Haskell.
Among the lifted functions from `XmlNode`, `ArrowXml` also provides additional arrows, for instance to process the whole attribute list. How to use these arrows for processing XML documents in "real-life"-programs will be described in the following chapter, where a simple RDF/XML parser is implemented.

State for XML Processing

The processing of XML documents also needs a state for global processing options, like encoding options, the document base URI, trace levels or error message handling. The

module `XmlIOStateArrow` provides this, based on `IOStateListArrow`. That is the list arrow implementation with state and I/O handling. The main data type defined in this module is `XIOS` which represents the global store and is defined as follows:

```
newtype XIOS
    = XIOS (AssocList String (IOSArrow XmlTree XmlTree))

type IOSArrow b c = IOSLA XIOS b c
```

Again, `AssocList` is used to provide a key-value list. But this time the value is not just a string but a state arrow for `XmlTree`. This enables to store – besides simple trace messages which are just character strings – whole XML trees for collecting error messages and functions, e.g. for error message handling. The type `IOSArrow` is an alias for state arrows with a `XIOS` state.
The main entry point for running a state arrow with I/O is the function `runX` and its type definition is:

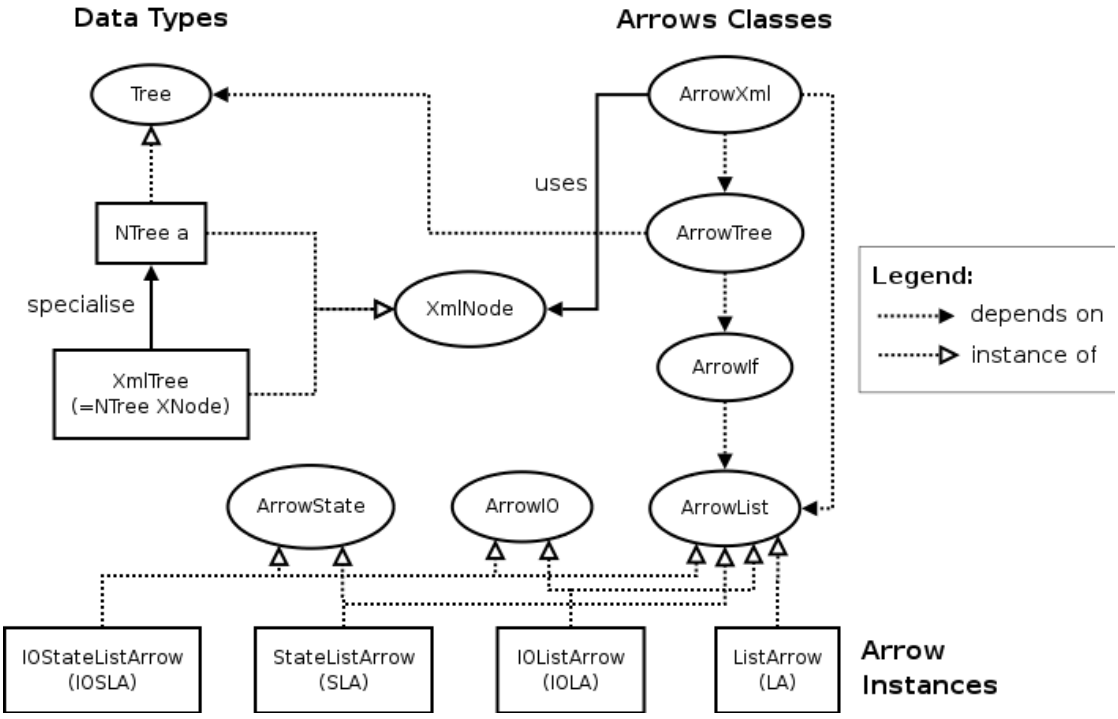
```
runX :: IOSArrow XmlTree c -> IO [c]
```

This function is used to start all kind of XML processing. While invoking `runX f`, an empty XML root node is applied to `f`. Usually, `f` will start with a constant arrow that ignores the input. This can be an arrow which reads in the XML document. The usage of `runX` is shown in examples later in this document.

2.4.7. Final Structure

The relationship between the different arrow classes, their instances and the data types of the Haskell XML Toolbox is illustrated in figure 2.1. It clearly shows, that the class `XmlNode` acts as the interface between the data type `XmlTree` and the class `ArrowXml`. The arrows provided by `ArrowTree` depend on the functions defined in the class `Tree`. The figure is limited to the most important classes and data types.

Figure 2.1: Arrow Class and Data Type Structure



3. Example RDF/XML Processing

3.1. Introduction

Now that the structure, the basic ideas as well as the different classes of the Haskell XML Toolbox have been described, this chapter will demonstrate how to design a program with it. The implementation of an RDF/XML parser was chosen as an example, because all parts of the Haskell XML Toolbox, from simple file handling to complex arrows, are needed for reaching this goal. But before starting the programming, the language and the idea of RDF need to be discussed.
The Resource Description Framework (RDF) is a language for representing information about resources in the World Wide Web [RDF Primer]. The main intention of RDF is to represent meta data about Web resources, e.g. the author, title, or the modification date of the Web page. However, RDF can also be used more generally to represent information about all kinds of things that can be identified on the Web. This can be information about the goods of an online shop or the description of a Web service provided by a Web page.
The idea of RDF is not to provide another concept of displaying information to people like the Hypertext Markup Language (HTML), but rather to make the information processable by applications. The handling of RDF is based on a common framework, so the information can be processed outside the environment in which it was created or even combined with other information.
In RDF, every resource can be identified by Web identifiers called Uniform Resource identifiers (URI). The resources are described in terms of simple properties and property values. This defines a simple but flexible data model which represents the resources as graphs of nodes and arcs.
The following sections give an overview of the concept of RDF based on the following three documents: Primer [RDF Primer], concepts [RDF Concepts] and syntax specification [RDF/XML Syntax].

3.1.1. Basic Concepts of RDF

The purpose of RDF is to describe resources in the Web. In other words, it is used to make statements about resources. As already mentioned, these statements are defined

as a thing identified by an URI and described in terms of a property with a value. In an RDF statement, these three things are named *subject*, *predicate* and *object*. Together they form a pattern which is called a *triple*. Like in natural languages, the subject is the thing which is described and identified, the predicate is the property of the subject and the object is the value of the predicate. An example of an English statement is:

http://www.example.org/index.html has a **creator** whose value is **John Smith**

The RDF terms of this statement are:

- the *subject* is the Uniform Resource Locater (URL) to identify the Web page http://www.example.org/index.html
- the *predicate* is the word "creator"
- the *object* is the phrase "John Smith"

This example shows how RDF is structured, but it is not complete. In RDF, everything is identifiable and ambiguity should be avoided to ensure that the statements are machine-processable. Ambiguity among subjects, predicates or objects would mean that they cannot be clearly identified and, therefore, not processed by an application. To solve this problem, RDF uses Uniform Resource Identifier (URI) [URIS] extended with references called URI reference or *URIref*.
In the example above, the subject, namely the Web page, is identified by an URL. But in order to identify every part of the statement, it is also necessary to describe aspects which have no network locations and URLs. URIs are more general and not limited to identifying things in the Web. URLs are, in fact, only a particular kind of URI.

To be machine-processable, RDF needs to be represented in a way that applications can read it. There are different languages to serialise RDF. One is *N-Triples*, which is often used to explain RDF and to write down simple statements because it is very short and clear. The other one is *RDF/XML*, the official language, recommended by the W3C. RDF/XML uses the Extensible Markup Language (XML), which allows to design own document formats and write documents in that format. A detailed description about XML can be found in the document [XML]. RDF/XML and its syntax will be described more thoroughly in section 3.1.3.

3.1.2. Model of RDF

The example statement of the last section:

http://www.example.org/index.html has a **creator** whose value is **John Smith**

can also be represented by the following triple:

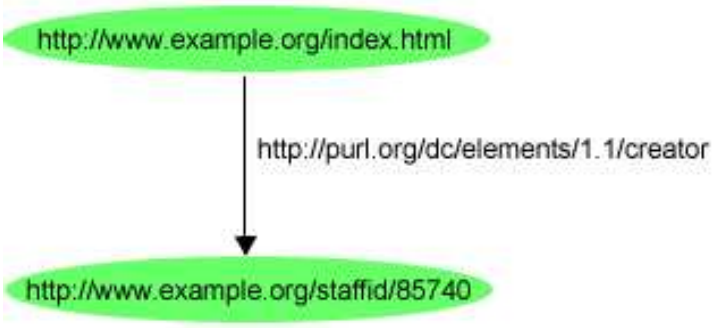
- the subject: `http://www.example.org/index.html`
- the predicate: `http://purl.org/dc/elements/1.1/creator`
- the object: `http://www.example.org/staffid/87540`

In RDF, neither the subject nor the predicate must be character strings. That is because every element of a triple should be identifiable and only the object can be a so-called *literal*. Those literals can be *plain literals*, which is a simple character string, or *typed literals*, where the literal has an additional data type information. Both, plain and typed literals, can contain Unicode characters that allow to represent information from many languages directly.
The URIrefs used in triples are often organised so that they define a vocabulary. The predicate URIref `http://purl.org/dc/elements/1.1/creator`, for example, is part of the Dublin Core vocabulary [DC]. In this vocabulary there are definied several commonly used terms, like the term *creator*. The advantage of vocabularies lies in their reusability. Applications using the same vocabulary understand or interpret the predicates of triples in the same way, since their meaning and their usage are defined. Vocabularies for RDF are written in the RDF Vocabulary Description Language [RDF Schema], which shall not be described here.
The statements of RDF can be modelled as nodes and arcs in a graph and in this notation a statement is represented by

- a node for the subject
- a node for the object
- a directed arc from the subject node to the object node for the predicate

So, the statement above can be modelled by the graph in figure 3.1.

Figure 3.1: Simple RDF Graph



Another way to represent or serialise RDF statements using English statements is the *N-Triples* notation, which is a fixed subset of N3 [N3]. The N-Triples notation requires that URI references are written out completely in angle brackets. This can result in very long lines per page. Therefore, in order to abbreviate full URI references, qualified names (see section 2.2.2) are used to avoid this. The following list includes the qualified names that are used throughout this document and without explicitly specifying them each time:

- prefix `rdf:` namespace URI: `http://www.w3.org/1999/02/22-rdf-syntax-ns#`
- prefix `dc:` namespace URI: `http://purl.org/dc/elements/1.1/`
- prefix `ex:` namespace URI: `http://www.example.org/`
- prefix `exterms:` namespace URI: `http://www.example.org/terms/`
- prefix `exstaff:` namespace URI: `http://www.example.org/staffid/`
- prefix `xsd:` namespace URI: `http://www.w3.org/2001/XMLSchema#`

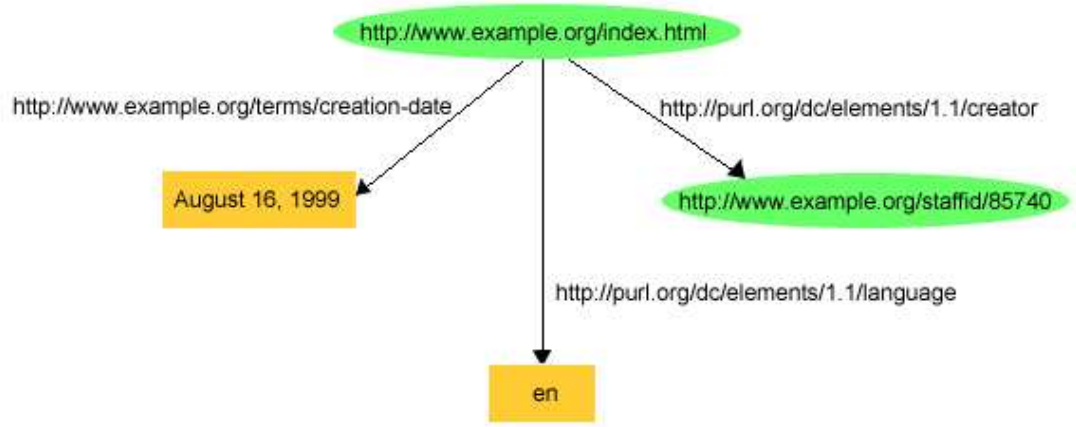
The next listing shows a group of statements with additional information added to the example above in N-Triples notation and using this new shorthand:

Listing 3.1: Group of Statements with N-Triple Notation

```
ex:index.html   dc:language             "en" .
ex:index.html   exterms:creation-date   "August 16, 1999" .
ex:index.html   dc:creator              exstaff:85740 .
```

The RDF graph of that group is shown in figure 3.2.

Figure 3.2: Compound RDF Graph



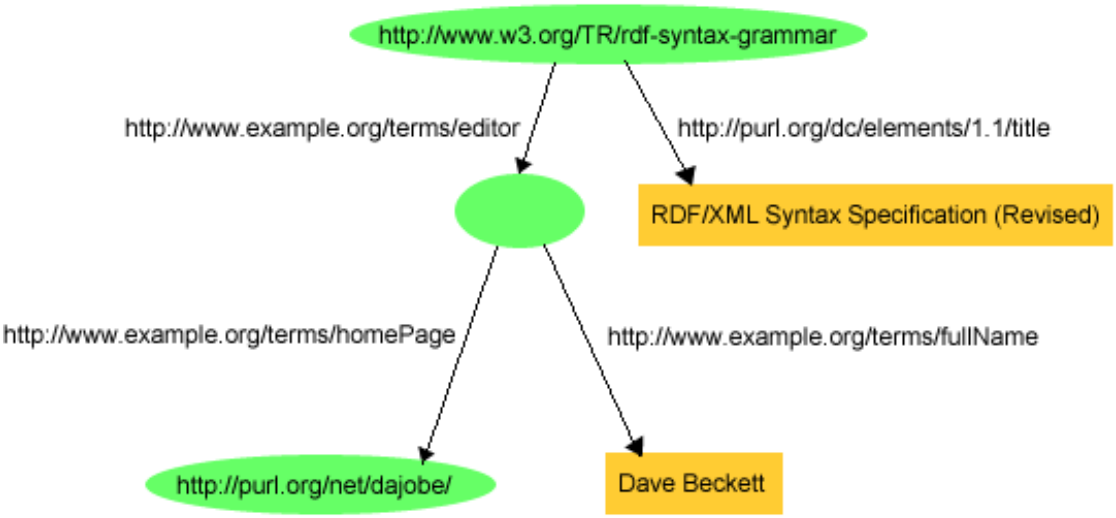
This example illustrates the use of vocabularies: An organisation such as example.org has a vocabulary consisting of URIrefs starting with the prefix `http://www.example.org/terms/` for terms it uses in its business, such as "creation-date" or "product". Another vocabulary of the organisation is to identify its employees with URIrefs starting with `http://www.example.org/staffid/`. RDF itself uses the same approach to define its own vocabulary of terms with special meanings in RDF. As shown in the list above, the URIrefs in the RDF vocabulary all begin with `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, which is conventionally associated with the QName prefix `rdf:`.
In addition to this, it is crucial to distinguish between any meaning that RDF for itself associates with terms such as `dc:creator` and externally-defined meaning that programs might associate with those terms. Only the semantics of the RDF vocabulary and the graph syntax are directly defined in the documents [RDF/XML Syntax] and RDF-Semantics [RDF Sematics]. The meaning of any other vocabularies is defined somewhere externally to RDF. Thus, generic RDF applications like parsers, would only recognise a statement with a specific vocabulary as a valid triple, but would not add and verify any special meaning that might be associated with any terms of the vocabulary.

Blank Nodes

Graphs may also include nodes without URIrefs, i.e. *blank nodes*. The idea behind blank nodes is to represent something that does not have an URIref, but can be described in terms of other information. As an example, figure 3.3 shows a graph which represents

the statement "the document 'http://www.w3.org/TR/rdf-syntax-grammar' has a title 'RDF/XML Syntax Specification (Revised)' and has an editor, the editor has a name 'Dave Beckett' and a home page 'http://purl.org/net/dajobe' ".

Figure 3.3: Graph with a Blank Node



The concept of the editor of a Web page does not need to be referred directly from outside a particular graph and hence, does not require a "universal" identifier. The blank node simply provides the necessary connectivity between the remaining parts of the graph. However, to represent this graph with the N-Triples notation, the blank nodes also need some form of explicit identifier. N-Triples use *blank node identifiers* having the form `_:name` for that, where `name` can be any kind of character string. In most cases, this is a number. For instance, a blank node identifier `_:id1` might be used to refer to the blank node. To keep this example short, only the subgraph with the blank node is shown:

Listing 3.2: Triples with Blank Nodes

```
<http://www.w3.org/TR/rdf-syntax-grammar>   exterms:editor   _:id1 .

_:id1   exterms:homePage    <http://purl.org/net/dajobe> .

_:id1   exterms:fullName    "Dave Beckett" .
```

Blank node identifiers are not actual parts of the RDF graph, like URIrefs and literals

are. They serve as representatives of the blank nodes in a graph and distinguish one blank node from another when the graph is written in the N-Triple form. Additionally, blank node identifiers also have significance only within the triples representing a *single* graph.
Because RDF can only represent *binary* relations directly, blank nodes also provide a way of dealing with n-ary relations by breaking them up into a group of separate binary relations.

Typed Literals

In some situations, it might be more appropriate to store some kind of type information to the plain literal of a statement. For example, if the age of a person is to be recorded, it does not make sense to use a plain literal for this, like illustrated in the next listing:

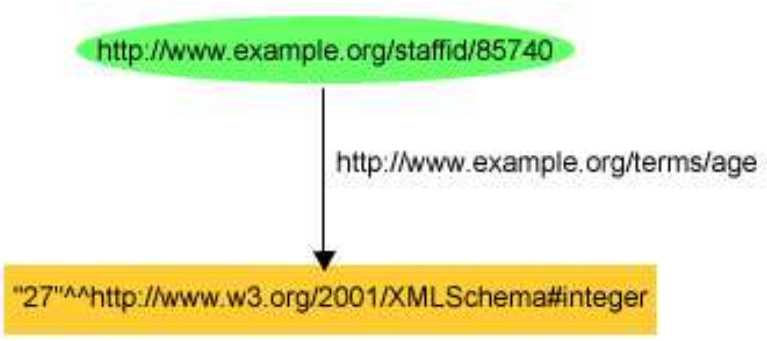
```
exstaff:85740    exterms:age      "27" .
```

An application processing this statement cannot "know" that the literal "27" is a decimal number and not a character string. Therefore, the data type information of this literal is added to the statement like it is done in database systems or programming languages. In RDF, these extended literals are called *type literals*. The example above can be rewritten using a typed literal to store the age:

```
exstaff:85740    exterms:age      "27"^^xsd:integer .
```

The same example is show as a graph in figure 3.4.

Figure 3.4: Graph with a Typed Literal



In this example the data type `integer` defined in the XML Schema [XML Schema] data types is used. It is very important to keep in mind that the data types of the typed literals are not validated by generic RDF applications. As it is the case with the vocabulary, the data type definition is done externally from the RDF definition, which means that an RDF parser cannot determine whether a literal is valid or not. This has to be done by the applications which know how to process these particular data types.

This section has presented an introduction to the concepts of RDF. It has shown how RDF can be used to store all kinds of data and that RDF graphs are similar to the way of recording information in simple relational databases.

3.1.3. RDF/XML - Syntax

Although the conceptual model of RDF is a graph, it is also necessary to write down and exchange or, in other words, to serialise RDF graphs. The N-Triples notation, which has been already introduced, can be used for this but it was only intended as a shorthand notation. RDF/XML instead, is the normative syntax for serialising RDF graphs and is defined in the RDF/XML Syntax Specification [RDF/XML Syntax]. The basic ideas behind the RDF/XML syntax can be illustrated by using some of the examples presented previously. The example in listing 3.3 has taken the first example and has replaced the URIref of the object by a plain literal.

Listing 3.3: RDF Statement with Plain Literal

```
ex:index.html    exterms:creator      "John Smith"
```

This RDF statement can be encoded in RDF/XML Syntax as the following listing:

Listing 3.4: RDF/XML for the Creator Concept

```
1 <?xml version="1.0"?>
2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3             xmlns:exterms="http://www.example.org/terms/">

5     <rdf:Description rdf:about="http://www.example.org/index.html">
6         <exterms:creator>John Smith</exterms:creator>
7     </rdf:Description>

9 </rdf:RDF>
```

Lines 1-3 are the "introduction" of the RDF/XML document. First, in line one, there is the XML declaration to indicate that the following content is XML. Then, in line two

and three, there begins an `rdf:RDF` element which indicates that the following content up to the end tag is RDF. The namespace declarations for the RDF namespace, defining the terms from the RDF vocabulary and all the others which are used throughout this document, are also written down in these lines. Although the RDF namespace and the `rdf:RDF` element are optional in situations where the XML can be identified as an RDF/XML document by context, it is always better to provide them, since this makes the documents much clearer.
In line five there is the `rdf:Description` element, which indicates the start of a *description* of a resource and is also called *node element*. What this statement is *about*, is specified in the `rdf:about` attribute. The content of this attribute is the URIref of the subject. The following line, line six, provides a *property element*, which represents the predicate and object of the statement. The qualified name `exterms:creator` of this element specifies the predicate and the content of the property element is the plain literal `John Smith`, which is the object.
The listing 3.4 illustrates the basic ideas of RDF/XML. There are several other ways of representing an RDF statement in XML and also a lot of abbreviations to keep the RDF/XML documents from becoming too big. All of these options are described in [RDF/XML Syntax] and this section introduces only those which are shown in [RDF Primer].
Making more than one statement about the same resource can be represented in RDF/XML by using the lines 5-7 in listing 3.4 for every new statement. To avoid this overhead of repeating the same `rdf:Description` element for every statement, RDF/XML allows multiple property elements, representing the predicates and objects for the same subject resource. For example, listing 3.5 represents the following group of statements:

```
ex:index.html   dc:creator              exstaff:85740
ex:index.html   exterms:creation-date   "August 16, 1999"
ex:index.html   dc:language             "en"
```

These triples are the same as the RDF graph shown in figure 3.2.

Listing 3.5: Multiple Properties

```
1 <?xml version="1.0"?>
2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3             xmlns:dc="http://purl.org/dc/elements/1.1/"
4             xmlns:exterms="http://www.example.org/terms/">

6   <rdf:Description rdf:about="http://www.example.org/index.html">
7     <exterms:creation-date>August 16, 1999</exterms:creation-date>
8     <dc:language>en</dc:language>
```

```
 9     <dc:creator rdf:resource="http://www.example.org/staffid/85740"/>
10   </rdf:Description>

12 </rdf:RDF>
```

Apart from showing how to use multiple property elements, the listing 3.5 also introduces a new form of property element. Unlike the first two properties, the property in line nine is an empty element with an `rdf:resource` attribute. This attribute represents a property whose value is another *resource*, rather than a literal. If the URIref of this resource has been written in the same way as the literal values, the URIref would not be a resource identifier but a character string. Unfortunately, the URIref cannot be abbreviated as a qualified name and has to be written out, since it is being used as an attribute value.
Using blank nodes to describe resources is another technique of RDF. Blank nodes are nodes which do not have an URIref but can be described in terms of other information as discussed in section 3.1.2. RDF/XML provides several ways to represent graphs containing those nodes, but only the most direct approach should be illustrated here. This is to assign a *blank node identifier* to each blank node. A blank node identifier only identifies a blank node within a particular RDF/XML document and is, in contrast to an URIref, unknown outside the document. In order to refer to a blank node in RDF/XML, the `rdf:nodeID` attribute and the identifier as its value are used. This attribute can appear in places where the URIref of a resource would otherwise be and replaces the `rdf:about` attribute of node elements (elements with the name `rdf:Description`). The example in listing 3.6 shows the RDF/XML corresponding to figure 3.3.

Listing 3.6: Blank Nodes in RDF/XML

```
 1 <?xml version="1.0"?>
 2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 3             xmlns:dc="http://purl.org/dc/elements/1.1/"
 4             xmlns:exterms="http://www.example.org/terms/">

 6   <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
 7     <dc:title>RDF/XML Syntax Specification (Revised)</dc:title>
 8     <exterms:editor rdf:nodeID="abc"/>
 9   </rdf:Description>

11   <rdf:Description rdf:nodeID="abc">
12     <exterms:fullName>Dave Beckett</exterms:fullName>
13     <exterms:homePage rdf:resource="http://purl.org/net/dajobe/"/>
14   </rdf:Description>

16 </rdf:RDF>
```

In listing 3.6 the value of the `exterms:editor` property is a blank node with the identifier `abc`. The blank node itself is defined in line 11 as the subject of two statements.

Instead of using plain literals as usual, the typed literals introduced in section 3.1.2 may be used as well. This can be done in RDF/XML by adding an `rdf:datatype` attribute with the data type URIref as its value to the property element. The following listing illustrates the graph shown in figure 3.4 in the RDF/XML syntax:

Listing 3.7: Typed Literal in RDF/XML

```
 1 <?xml version="1.0"?>
 2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 3             xmlns:exterms="http://www.example.org/terms/">

 5   <rdf:Description rdf:about="http://www.example.org/staffid/85740">
 6     <exterms:age rdf:datatype=
 7       "http://www.w3.org/2001/XMLSchema#integer">27
 8     </exterms:age>
 9   </rdf:Description>

11 </rdf:RDF>
```

The RDF/XML Syntax described so far provides a simple but general way of expressing RDF graphs. This simple approach gives the most direct representation of the actual graph structure. Some other additional abbreviations provided by RDF/XML will be introduced in section 3.2 along with the example RDF/XML parser.

3.1.4. SPARQL – Query Language for RDF

The approach of storing information in RDF is very similar to that of simple relational databases. Almost every relational database provides the Structured Query Language (SQL) to access the information recorded in the tables of a database. SQL provides several commands to manipulate and to search for data. It seems likely to provide such a data access language also for RDF, which could be used to query the information stored in RDF graphs. There are several approaches of query languages for RDF and one of them should be introduced here. This is the query language part of the *Protocol and RDF Query Language* (SPARQL) [SPARQL] which is still under development and not finished yet. SPARQL provides a lot of features to search for information in RDF, like querying more than one graph and connecting several RDF stores. As this document intends to exemplify how to implement an RDF/XML parser with a small query engine, only the basic language features of SPARQL should be introduced here.

The idea of the SPARQL query language is based on matching *graph patterns*. There are several versions of graph patterns. The simplest one is the *triple pattern* which is the same as an RDF triple but with the possibility of a variable in any of the subject, predicate or object positions. An exact match of this triple pattern to an RDF graph is needed to fulfill a pattern. A simple query which searches for the title of a book in the following data:

```
ex:book/book1 dc:title "SPARQL Query"
```

shows the next listing:

Listing 3.8: Simple SPARQL Query

```
SELECT ?title
WHERE { <http://www.example.org/book/book1>
        <http://purl.org/dc/elements/1.1/title>
        ?title . }
```

The result of this query would be:

title
"SPARQL Query"

A query always consists of two parts, the `SELECT` clause and the `WHERE` clause. The `SELECT` clause specifies the variables, which should be returned by the query. The `WHERE` clause contains the graph pattern. In this case the graph pattern consists one single triple pattern. Graph patterns can also be of other pattern types. Every pattern adds additional functionality to SPARQL. These additional pattern types will not be taken into consideration.
The triple pattern consists of three query terms. These can be an URIref delimited by "<>". The object can also be a literal which is delimited by single or double quotes and can be followed by an optional language tag, introduced with '@', or an optional data type URI, introduced by '^^'. As a convenience, integers, floating point numbers and boolean values can be directly written and are interpreted as typed literals of data type `xsd:integer`, `xsd:double` or `xsd:boolean`.
The variables used in a SPARQL query have a global scope and are indicated by '?' or '$'.
The term "binding" is used as a descriptive term to refer to a pair of variable and RDF term. These bindings result in tabular form so if variable x is bound to "Fred" it is shown as:

x
"Fred"

The results of a query can also be returned in RDF, in a special XML format defined in [SPARQL Result] or in forms specific to the implementation.
The matching of a graph pattern to a graph, gives bindings between the variables of the triple patterns and the RDF terms. This means, the triple pattern, with the variables replaced by the corresponding RDF terms, is a triple of the graph being matched. This process is also called *substitution*.
The last example has shown how to search for only one triple. In the following example, the data consists of two triples with blank nodes as the subject and the query also consists of two triple patterns. This graph pattern only matches if *each* of the triple patterns matches with the *same* substitution. The data is:

```
_:a dc:name "Johnny" .
_:a dc:mbox <mailto:jlow@example.com> .
```

The query to search for the mail address is:

```
SELECT ?mbox
WHERE { ?x <http://purl.org/dc/elements/1.1/name> "Johnny" .
        ?x <http://purl.org/dc/elements/1.1/mbox> ?mbox }
```

This leads to the following result:

mbox
<mailto:jlow@example.com>

With SPARQL, it is also possible to deal with multiple matches. The example above could have two mail addresses as a result if the data contained two entries with the name "Johnny".
Triple patterns are written as a list of subject, predicate and object, which can result in very long lists and can therefore be abbreviated. The *predicate-object list* allows to write down the subject only once where several triple patterns with the same subject are needed. The predicate-object list is then delimited by a ';' like shown in the next listing:

```
?x  dc:creator          ?name ;
    dc:creation-date     ?date .
```

Another way of abbreviating triple patterns are *object lists*, which can be used when triple patterns share both subject and predicate. The following listing illustrates how

the object list is used.

```
?x  dc:creator  "Alice" , "Alice_" .
```

This is the same as writing the triple patterns in the following form:

```
?x  dc:creator  "Alice" .
?x  dc:creator  "Alice_" .
```

Like section 3.1.3, this section illustrated some, but not all of the abbreviations and language features of the SPARQL query language. A number of additional ones will be described in further detail in the next sections, but the language shown so far already provides enough features to write simple and useful queries for RDF triples.

3.2. Writing the RDF/XML Parser

3.2.1. Introduction

The RDF/XML parser, which shall now be implemented, is structured and based on the complexity of the tasks. First of all, a function or module is needed which takes care of the I/O handling, i.e. reading an RDF document and presenting the result of the parsing process. Then, the real processing of the document needs to be implemented, which is also separated in several parts. The first part is to normalise all the RDF/XML abbreviations into a simple and normative RDF/XML document. This normalised document is then processed and the result is stored in a new data structure. After that, the data structure can be printed out in several formats like the N-Triples notation or a simple RDF/XML representation. Furthermore, a SPARQL parser and a module for searching in the RDF data structure is also a part of it, in order to provide simple query features.

3.2.2. Main Function and Option Handling

The first part of a program is the main function, which is the "main entry point". The file handling and the additional processing of the XML document are coordinated in this function. The Haskell XML Toolbox provides – besides the arrows introduced in section 2.4 – the module `XmlStateFilterInterface`, which consists of compound arrows for reading, parsing, validating and writing XML documents. The most important arrows of this module are `readDocument` and `writeDocument`. The arrow `readDocument` is the main document input arrow. It can be configured by an option list of type `Attributes`

and takes the file name of the document which should be read as parameter. The attribute list of `readDocument` is stored in the global state component to ensure that it can be accessed everywhere in the program. Some of the available options are:

- `a_validate` : validate document, else skip validation (default)
- `a_check_namespaces` : check namespaces, else skip namespace processing (default)
- `a_canonicalize` : canonicalise document (default), else skip canonicalisation
- `a_remove_whitespace` : remove all whitespace, used for document indentation, else skip this step (default)
- `a_trace` : trace level: values: 0 - 4

The writing of XML documents to files is done by the arrow `writeDocument`. Like `readDocument`, it takes a list of options and the file name of the document which should be written. The full list of the available options for both arrows is listed in appendix A. The next listing, for example, illustrates how to write a main function that reads an RDF file "simple.rdf" in, processes all namespaces, removes all white spaces and then creates a new RDF file "output.rdf" out of it:

Listing 3.9: First Main Function

```
main :: IO ()
main
  = do
    runX ( readDocument [(a_check_namespaces,"1"),
                         (a_remove_whitespace,"1")]
                        "simple.rdf"
           >>>
           writeDocument [] "output.rdf"
         )
    return ()
```

Unlike the two options handed over to `readDocument`, the arrows `removeAllWhiteSpace` and `propagateNamespaces` can be used to delete the whitespaces and process all namespaces. Line-breaks to indent a XML document are interpreted as XML text nodes. When processing those documents, it is crucial to call `removeAllWhiteSpace`, since these extra text nodes are also returned by `getChildren`. Sometimes, the number of chil-nodes of an element are of interest and text nodes containing line-breaks disturb or cover up the real number which maybe leads to incorrect results.

The function `runX`, described in section 2.4.6, is used to run the processing. In this case, it consists of the two arrows `readDocument` and `writeDocument` combined with the operator `(>>>)`, which results in an I/O action. This action is executed by `runX` and the main function ends by returning the empty tuple. Several errors, like XML syntax error, can occur during the processing. These errors, however, are ignored and the main function always ends successfully.
To enable the error handling in the example above, the arrow `getErrStatus` from the module `XmlIOStateArrow` reads the error status from the global state, which can then be used to determine, how the main function should end. The example in listing 3.10 shows how the main function can be rewritten, so as to enable error handling and tracing parse errors while processing.

Listing 3.10: Main Function with Error Handling

```
main
  = do
    [rc] <- runX ( readDocument [(a_trace,"1")]
                                "simple.rdf"
                   >>>
                   removeAllWhiteSpace >>> propagateNamespaces
                   >>>
                   writeDocument [] "output.rdf"
                   >>>
                   getErrStatus
                 )
    exitWith ( if rc >= c_err
               then ExitFailure (-1)
               else ExitSuccess )
```

When including the main function from the example above into a Haskell module, two modules have to be imported. One of them is the module `Text.XML.HXT.Arrow` which is the application programming interface to the arrow modules of the Haskell XML Toolbox and exports all important arrows, basic data types and functions. The other one is the module `System.Exit` from the standard Haskell libraries which is used to get the functions for advanced program termination.
However, the example above is still not completed. Another considerable feature which should be provided by every program is to deal with command-line options. Furthermore, the name of the input and output file should not be hard coded but rather specified by a command-line argument. Therefore, the main function has to be rewritten again.

Listing 3.11: Main Function with Commandline Options

```
main
    = do
      argv      <- getArgs
      (al, src) <- cmdlineOpts argv
      [rc]      <- runX (processDocument al src)
      exitWith ( if rc >= c_err
                 then ExitFailure (-1)
                 else ExitSuccess )
```

Listing 3.11 shows that, the command-line parameters are collected first by the Haskell function `getArgs`. After that, the function `cmdlineOpts` processes these parameters and returns a pair of the list of processed arguments and the input file name. This pair, then, is handed over to the arrow `processDocument` which is executed by `runX`. Finally, the error handling is done in the same way as shown in the last example.
The function `cmdlineOpts` parses the command-line parameters, prints the list of available command-line arguments or terminates the program if no input file name has been specified. The definition of it shall not be described here in detail, because it is not a Haskell XML Toolbox specific function, but a standard Haskell code.
The main function is much shorter now that the processing arrows are encapsulated by the arrow `processDocument`, which is defined as follows:

Listing 3.12: processDocument

```
processDocument :: Attributes -> String -> IOSArrow b Int
processDocument al qr src
    = readDocument al src
      >>>
      removeAllWhiteSpace >>> propagateNamespaces
      >>>
      writeDocument al (fromMaybe "" (lookup a_output_file al))
      >>>
      getErrStatus
```

The arrow takes the attributes and the name of the input file as extra parameters. These arguments are handed over to `readDocument`. Moreover, the command-line option for the name of the output file is looked up in the list of parameters `al`. If this parameter does not exist, the output is printed to the standard output. This list can contain any options, allowed by `readDocument` and `writeDocument`. Later in this section, an extra command-line option will be added to the available list of arguments.

The program described so far is already a fully working application, which parses a XML document and writes it back to a file or the standard output. However, it does not contain any RDF/XML specific arrows yet. The arrow `processDocument` can be extended easily by any arrow which processed XML somehow. In the next section this will be shown by implementing parser arrows for RDF/XML.

3.2.3. Parsing RDF/XML

Before starting to parse the full RDF/XML syntax, only a subset of it is processed. This subset covers all the features described in section 3.1.3. The additional syntax abbreviations and their processing are discussed in the next section.

Simple Processing Arrows

The first simple RDF/XML document which should be processed by the parser is the document presented in listing 3.4. The next listing shows this RDF document in order to recall the syntax of RDF/XML (the XML and namespace declaration have been removed):

```
<rdf:Description rdf:about="http://www.example.org/index.html">
    <exterms:creator>John Smith</exterms:creator>
</rdf:Description>
```

It contains no abbreviations and has only one RDF triple with a plain literal as the object. First of all, the processing functions return their results in a simple XML document and later it will be shown how to store these results in a special data structure for RDF. The structure of this XML document is illustrated in the next listing, using the data of listing 3.4:

Listing 3.13: Simple Triple Representation

```
<triple>
    <subject>http://www.example.org/index.html</subject>
    <predicate>http://www.example.org/terms/creator</predicate>
    <object>John Smith</object>
</triple>
```

To achieve this result, every `rdf:Description`-element has to be selected by processing the whole XML document tree. The following listing shows the predicate, which tests whether an element has the name `rdf:Description` and the attribute `rdf:about`.

Listing 3.14: Predicate detecting Node Elements

```
isNodeElem :: (ArrowXml a) ⇒ a XmlTree XmlTree
isDesc
  = isElem >>> hasQName rdf_Description >>> hasQAttr rdf_about
```

The constants `rdf_Description` and `rdf_about` are of type `QName` and represent the qualified names.
This predicate can now be applied to the whole document tree. If the predicate holds, this element can be further processed. The traversing through the tree is done by the arrow `deep`:

Listing 3.15: Apply the Predicate to the Tree

```
processRDF :: (ArrowXml a) ⇒ a XmlTree XmlTree
processRDF
  =  processChildren (deep (isNodeElem `guards` getTriple))
```

The arrow `getTriple` does the actual processing of the `rdf:Description` elements. It selects the value of the `rdf:about` attribute and creates an element with the name "subject" and the value as a text element. The same is done with the property element by selecting the predicate and object value and generating elements with the name "predicate" and "object" out of it. These three elements are then added to the list of children of an element called "triple":

Listing 3.16: getTriple

```
getTriple :: (ArrowXml a) ⇒ a XmlTree XmlTree
getTriple
  = selem "triple"
    [getQAttrValue rdf_about >>> selem "subject" [mkText]
    ,getChildren >>> getUniversalUri
                 >>> selem "predicate" [mkText]
    ,selem "object" [getChildren>>>getChildren]
    ]
```

The arrow `processRDF` can now be added to the `processDocument` arrow from listing 3.12, so that it processes the XML tree generated by `readDocument` and passes the result of it to the arrow `writeDocument`.
The resulting program is already able to process very simple RDF documents with one or more triples, but every triple has to be declared explicitly and no abbreviations are

allowed. In order to process the multiple property elements as shown in listing 3.5, the function above has to be rewritten, because the URIref of the subject is needed several times. The function `getTriple` now takes an additional parameter of type string which is the URIref of the subject to ensure that it can be used more than once. The listing 3.17 presents the rewritten function `getTriple`.

Listing 3.17: getTriple for Multiple Properties

```
getTriple subject
  = getChildren >>> selem "triple"
    [selem "subject" [txt subject]
    , getUniversalUri >>> selem "predicate" [mkText]
    , selem "object" [getChildren]
    ]
```

Furthermore, a new arrow is needed to select the value of the `rdf:about` attribute. This new arrow is called processSubject and is defined as follows:

```
processSubject :: (ArrowXml a) ⇒ a XmlTree XmlTree
processSubject = arr getTriple $< getQAttrValue rdf_about
```

It hands the value of the attribute over to the arrow `getTriple` via the combinator `$<`, which allows the programmer to use a parameter more than once (see section 2.4.1). This arrow shows, that the special combinators for point-wise programming are highly practical. Without them the code of such a simple arrow would be awkward and confusing.
Now, `processSubject` and not `getTriple` has to be applied to the whole tree in the arrow `processDocument`. Every time the predicate `isNodeElem` holds, the `rdf:about` value is selected and handed over to `getTriple`, which generates a triple element for every child-node of the `rdf:Description` element.

The next RDF/XML syntax feature that should be supported by the parser are blank nodes. There are several ways of representing blank nodes in RDF/XML, but first only blank nodes with a explicit blank node identifier should be processed. Therefore, the predicate `isNodeElem` has to be rewritten, because an `rdf:Description` element can also have a `rdf:nodeID` attribute carrying the blank node identifier instead of the `rdf:about` attribute. The choice between these two possibilities is made by the combinator `orElse`, as the next listing shows:

Listing 3.18: isNodeElem with Blank Node Test

```
isNodeElem = isElem >>> hasQName rdf_Description
         >>> (hasQAttr rdf_nodeID `orElse` hasQAttr rdf_about)
```

In addition to `isNodeElem`, the arrow `processSubject` also has to be rewritten, since the blank node identifier has to be selected as well as the URIref. The following listing illustrates how the choice between attributes can be made by the `choiceA`-construction (the type signature has been removed):

Listing 3.19: processSubject with Blank Node

```
processSubject
  = arr getTriple
    $< choiceA [hasQAttr rdf_about :-> getQAttrValue rdf_about
               ,hasQAttr rdf_nodeID:-> getQAttrValue rdf_nodeID
               ]
```

Moreover, the processing of the object of a triple has to be changed, since an object can also have an `rdf:nodeID` attribute referring to a blank node or an `rdf:resource` with an URIref. This is done by the arrow `processObject` which also uses `choiceA` to deal with the different cases:

```
processObject :: (ArrowXml a) ⇒ a XmlTree XmlTree
processObject
  = choiceA [isResource     :-> getQAttrValue rdf_resource
            ,isBlankNodeRef :-> getQAttrValue rdf_nodeID
            ,this           :-> getChildren >>> getText
            ]
    >>>
    selem "object" [mkText]
```

The predicates `isResource` and `isBlankNodeRef` test if the element is empty and has the right attribute. Now, the creation of the "object" element can be replaced in `getTriple` by `processObject` and the application is able to process RDF documents with blank nodes like the one in listing 3.6, which would generate the following output:

```
<triple>
  <subject>http://www.w3.org/TR/rdf-syntax-grammar</subject>
  <predicate>http://purl.org/dc/elements/1.1/title</predicate>
  <object>RDF/XML Syntax Specification (Revised)</object>
</triple>
<triple>
  <subject>http://www.w3.org/TR/rdf-syntax-grammar</subject>
  <predicate>http://www.example.org/terms/editor</predicate>
  <object>abc</object>
</triple>
```

```
<triple>
  <subject>abc</subject>
  <predicate>http://www.example.org/terms/fullName</predicate>
  <object>Dave Beckett</object>
</triple>
<triple>
  <subject>abc</subject>
  <predicate>http://www.example.org/terms/homePage</predicate>
  <object>http://purl.org/net/dajobe/</object>
</triple>
```

Finally, the typed literals need to be processed to support all the functionality of RDF/XML as presented in section 3.1.3. This time, however, only the arrow `processObject` has to be rewritten by adding a new case to the `choiceA`-construction. If an object is a typed literal, the value of the `rdf:datatype` attribute and the text of the element has to be selected as illustrated by the following listing:

```
processObject
  = choiceA [isResource     :-> getQAttrValue rdf_resource
            ,isBlankNodeRef :-> getQAttrValue rdf_nodeID
            ,isTypedLiteral :-> op1
            ,this           :-> getChildren >>> getText
            ]
    >>>
    selem "object" [mkText]

    where
    op1 = (getChildren >>> getText) &&& getQAttrValue rdf_datatype
          >>> arr ( λ(t,qn) = t++"^^"++qn)
```

The locally defined arrow `op1` selects the text of the property node and the URIref of the datatype attribute simultaneously. The resulting tuple is handed over to the lambda expression that combines the two character strings. Then, the lambda expression is converted to an arrow.
The language features of RDF/XML which can be processed by the application so far, provide sufficient ways of expression to represent every possible RDF graph. The next step in the development of the RDF/XML parser is to design a data structure in which the result of the processing should be stored without using a XML document for this.

Data Structure

An RDF triple consists of a subject, a predicate and an object. These three parts are represented by data types as follows:

Listing 3.20: Data Types Subject, Predicate and Object

```
type Subject = RDFTerm

type Predicate = URI

type Object = RDFTerm
```

The definition of the data types `RDFTerm` and `URI` are shown in listing 3.21

Listing 3.21: Types RDFTerm and URI

```
data RDFTerm = URIref URI
             | RDFLiteral String
             | RDFLangLiteral String String
             | RDFTypedLiteral String URI
             | BlankNode String
               deriving (Ord,Eq)

type URI = String
```

The constructors of `RDFTerm` express all the different units, a subject or an object can be. Actually, a subject can only be an URIref or a blank node and not the different kinds of literals. Later in this document, it will be shown that a subject can be a literal although this is forbidden by the RDF/XML Syntax specification. Besides typed and plain literal an object can also be a literal with a language specification, which has not been introduced yet.
The type URI is a character string to prevent the examples from becoming too complex. The following listing shows the definition of the data type representing a triple. The type `RDFStore` is a shortcut for a list of triples.

Listing 3.22: Triple Data Type

```
data Triple = Triple Subject Predicate Object
              deriving (Ord,Eq)

type RDFStore = [Triple]
```

The implementation of the `Show` class for all types, which enables data types to be printed out, is based on the N-Triples syntax and shall not be further described here. Furthermore, every data type has several constructor functions to create the different kinds of types.

Storing Parse-Result in the Data Structure

The output of every arrow has to be changed to the data types described above and unlike of creating XML elements, they have to use one of the constructor functions. Thus, `processObject` has to be rewritten as follows:

```
processObject :: (ArrowXml a) ⇒ a XmlTree Object
processObject
  = choiceA
    [isResource     :-> (getQAttrValue rdf_resource
                         >>> mkResourceA)
    ,isBlankNodeRef :-> (getQAttrValue rdf_nodeID
                         >>> mkObjectBlankNodeA)
    ,isTypedLiteral :-> op1
    ,this           :-> (getChildren
                         >>> getText >>> mkLiteralA)
    ]
    where
    op1 = getQAttrValue rdf_datatype &&& (getChildren >>> getText)
          >>> arr2 mkTypedLiteral
```

The arrow version of one of the type constructors is used, in every case of the choiceA-construct. The arrow `getTriple` has to be rewritten, too. It takes a `Subject` as extra parameter in place of the character string and has the type `Triple` as output, which is generated by the function `mkTriple`:

```
getTriple :: (ArrowXml a) ⇒ Subject -> a XmlTree Triple
getTriple subject
  = getChildren
    >>> (processPredicate &&& processObject)
    >>> arr2 (mkTriple subject)
    where
    processPredicate = (getUniversalUri >>> mkPredicateA)
```

The arrow `processSubject`, which calls `getTriple`, has the output type `Triple` as well and creates a `Subject` in the two cases of an URIref or a blank node at the

`rdf:Description` element. Furthermore, the arrow `getChildren` is applied in front of it, because the processing of the whole XML tree has changed, which will be shown later. The next listing illustrates the new version of `processSubject`:

```
processSubject :: (ArrowXml a) ⇒ a XmlTree Triple
processSubject
  = getChildren
    >>> (isNodeElem `guards` (arr getTriple $< blankOrNot))
    where
    blankOrNot = choiceA
      [hasQAttr rdf_about :-> (getQAttrValue rdf_about
                               >>> mkURIrefA)
      ,hasQAttr rdf_nodeID :-> (getQAttrValue rdf_nodeID
                                >>> mkSubjectBlankNodeA)
      ]
```

At last, the traversing of the tree and the types in `processRDF` have to be changed. The idea of the parser is to generate a list of triples out of the RDF/XML document. Accordingly, the triples created by `processSubject` have to be collected in a list. A list arrow with the input type `XmlTree` and a list of triples (i.e. `RDFStore`) as output type would provide exactly this functionality. The ensuing listing presents such a list arrow:

Listing 3.23: Collect Triples

```
triples :: XmlTree -> RDFStore
triples = runLA  ( processSubject)
```

This function can be used to rewrite `processRDF`, which also tests if the RDF/XML elements are encapsulated by a `rdf:RDF`-element where all namespaces are defined. Furthermore, the traversing of the tree has changed by a call of `getChildren` instead of the arrow `deep`:

```
processRDF :: (ArrowXml a) ⇒ a XmlTree RDFStore
processRDF = getChildren >>> isRDF `guards` arr triples
```

The function `processDocument` from listing 3.12 would stop working, if the arrow above were placed between the reading and the writing arrows. One reason is that `processRDF` is not of the correct type; its result is a `RDFStore` and not a `XmlTree`. This can be solved by an arrow that calls the show function of `RDFStore` and then creates a text node for every character string. But this would not be sufficient. The generated text nodes containing the result of the show function would not have a root node which is compellingly necessary for a `XmlTree`. The arrow `replaceChildren` solves this problem by replacing

the child elements of the root node with the text nodes containing the results. The next listing shows the line of code which has to be placed between `readDocument` and `writeDocument` to make `processDocument` successfully operate again:

```
replaceChildren (processRDF >>> (arr showRDFStore ) >>> mkText)
```

Since `processRDF` now creates a list of triples, it can be combined with all kind of arrows processing this list and cannot only be printed out. But before explaining this with a simply query language which makes it possible to search the list of triples for statements, the next section shows how more complex abbreviations of RDF/XML can be processed by the parser described so far.

3.2.4. Normalisation of Advanced RDF/XML Syntax Abbreviation

The idea behind the normalisation is to generate a simple standardised RDF/XML document out of a complex one that contains different abbreviations. The resulting document of the normalisation is passed to the arrow `processRDF`, which creates a list of parsed triples. This procedure keeps the actual parsing simple. Every additional abbreviations which have not been explained in section 3.1.3 will be introduced shortly, before the way how to cut them down is shown. The abbreviations are processed sequentially, which means that the whole document tree is completely extended several times and might result in a poor performance when parsing large RDF/XML documents. Nevertheless, this approach keeps the normalisation arrows simple and clearly focused on one problem to make them understandable and compact. In order to gain performance, they can be combined effortlessly, so that several abbreviations are processed during one tree walking-through.
All normalisation arrows are joined together in one arrow which generates the final result, as it has been done with `processRDF`.

Omitting Blank Node Identifiers and Nested Node Elements

So far, blank nodes have been represented by node elements with specific blank node identifiers. But it is also possible to omit these identifiers. Additionally, property elements which have a reference to a blank node, can contain a nested node element instead of referring to the blank node via a `rdf:nodeID` attribute. So, the RDF graph illustrated in figure 3.3 can be written down much shorter, as the following example shows:

Listing 3.24: Blank Nodes without Identifiers

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
            xmlns:dc="http://purl.org/dc/elements/1.1/"
            xmlns:exterms="http://www.example.org/terms/">
  <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
    <dc:title>RDF/XML Syntax Specification (Revised)</dc:title>
    <exterms:editor>
      <rdf:Description>
        <exterms:fullName>Dave Beckett</exterms:fullName>
        <exterms:homePage rdf:resource="http://purl.org/net/dajobe/"/>
      </rdf:Description>
    </exterms:editor>
  </rdf:Description>
</rdf:RDF>
```

Although the RDF/XML in example 3.6 is equivalent to the listing above and represents the same graph, the RDF parser described up till now is not able to process the last example. This is because, firstly, every node element is expected to have either a `rdf:about` attribute or a `rdf:nodeID` attribute and secondly, nested node elements are not allowed inside property elements. Processing listing 3.24 directly would be a lot more complicated than changing the XML tree in a way that the RDF parser can handle it. Hence, three arrows are needed: one to generate `rdf:nodeID` attributes where they are missing, one to break the nested node elements up and one to add the generated blank node identifier to the property element. The first arrow is `generateNodeID` which creates a blank node identifier for every node element. Its definition is shown in the next listing:

```
generateNodeID :: IOSArrow XmlTree XmlTree
generateNodeID
  = processTopDown (
      addAttrl (qattr rdf_nodeID attrValue)
      `when`
      (isElem >>> hasQName rdf_Description
       >>> neg (hasQAttr rdf_nodeID `orElse` hasQAttr rdf_about))
    where attrValue
            = getCounter "node_id" >>> arr("genid"++) >>> mkText
```

The additional attribute is added by the arrow `addAttrl`. It takes an arrow which generates a list of attributes and adds them to the existing list. In this case, the arrow `qattr` is used to create the new attribute. The value of it is defined locally in `attrValue`.

The arrow `getCounter` creates an integer value with the name "node_id" in the global state and returns the initialisation value or increments the global state by one and returns the new value if the state with the given name already exists. This function will be used several times to create unambiguous numbers. As illustrated in the next listing, the operator to use the output of an arrow more then once, is applied. It hands the integer value of the state over to the local defined function, which increments it and then converts it into a character string.

```
getCounter :: String -> IOSArrow b String
getCounter name
  = arr genNewId $< getParamInt 1 name
    where
    genNewId :: Int -> IOSArrow b String
    genNewId i
         = setParamInt name (i+1)
           >>>
           constA (show i)
```

The next step in the normalisation of blank nodes without identifiers and nested node elements is to convert the document into the tree structure, which the parser expects. Thus, every nested node element has to become a child-node of the root element. This can be achieved effortlessly by copying every existing node element in the document to the root element. Unfortunately, the tree contains duplicate node elements after this process. Moreover, the property elements that refer to the nested node elements still do not have the extra attribute with the URIref or the identifier of the `rdf:Description` element.
The copying of every node element to the root node can be done by the following expression:

```
processChildren (multi isNodeElem)
```

The arrow `multi` selects all subtrees for which the predicate holds and `processChildren` substitutes the children with this result. The listing 3.25 shows the intermediate result of normalising the document presented in example 3.24 (the namespace declarations have been removed).

Listing 3.25: Intermediate Result of Normalisation

```
1 <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
2   <dc:title>RDF/XML Syntax Specification (Revised)</dc:title>
3   <exterms:editor>
4     <rdf:Description rdf:nodeID="genid1">
```

```
 5       <exterms:fullName>Dave Beckett</exterms:fullName>
 6       <exterms:homePage rdf:resource="http://purl.org/net/dajobe/"/>
 7     </rdf:Description>
 8   </exterms:editor>
 9 </rdf:Description>
10 <rdf:Description rdf:nodeID="genid1">
11   <exterms:fullName>Dave Beckett</exterms:fullName>
12   <exterms:homePage rdf:resource="http://purl.org/net/dajobe/"/>
13 </rdf:Description>
```

The nested element, illustrated in line four to seven, has to be deleted, to ensure that the document after the normalisation is equal to the source document. Furthermore, the property element in line three has to be extended by the `rdf:nodeID` attribute and the value of the same attribute in line four. This can also be done by one arrow which is presented in the next listing:

```
deleteDuplicate :: (ArrowXml a) ⇒ a XmlTree XmlTree
deleteDuplicate
  = processTopDown (
     (processChildren(
       (addAttrl (getChildren >>> getAttrl) >>> setChildren [])
       `when`
       (getChildren >>> isNodeElem))
     )
     `when`
     (isNodeElem >>> (deep isNodeElem)))
```

First, the arrow searches for `rdf:Description` elements containing nested node elements. If one is found, the child-nodes, which are the property elements, are processed. Thereby, the attribute list of the nested node element is copied to the property element and the list of child-nodes is deleted.
Every normalisation step, which deals with omitted blank node identifiers and nested node elements, is now complete and can be sequenced in the arrow `normaliseRDF`:

```
normaliseRDF :: IOSArrow XmlTree XmlTree
normaliseRDF = seqA [generateNodeID
                    ,processChildren (multi isNodeElem)
                    ,deleteDuplicate
                    ]
```

The next example shows the final result of the normalisation, where the nested node elements have been deleted:

Listing 3.26: Final Result of Normalisation

```
1 <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
2   <dc:title>RDF/XML Syntax Specification (Revised)</dc:title>
3   <exterms:editor rdf:nodeID="genid1"/>
4 </rdf:Description>
5 <rdf:Description rdf:nodeID="genid1">
6   <exterms:fullName>Dave Beckett</exterms:fullName>
7   <exterms:homePage rdf:resource="http://purl.org/net/dajobe/"/>
8 </rdf:Description>
```

The parser which has been developed in the last section, is able to process this document and is therefore – in combination with the normalisation – also able to parse more advanced language features without actually extending the parser.

Unicode Support

In RDF/XML, every application of string refers to an Unicode character string, while non US-ASCII characters like 'ü' have to be replaced by the encoded version of the character, i.e. "\u00FC" (see [CHARMOD] for more information on encoding). Every attribute as well as every text node needs replacing. Again, this is a typical normalisation by processing the whole tree and converting all occurrences of strings. First, a function is needed replacing every non US-ASCII character by the correct encoded one, then this function has to be applied to the whole document tree.

```
stringToUtf :: String -> String
stringToUtf = concatMap charToUtf
    where
    charToUtf :: Char -> String
    charToUtf c
        | ord c < 0x80  = [c]
        | otherwise = "\\u00" ++charToHexString c
```

The function `charToHexString` is defined in the module `Text.XML.HXT.DOM.Util` which provides several utility functions; the one used here converts a character into a two-digit hexadecimal string.
Now, `stringToUtf` can be applied to all attributes and elements by `processTopDown-WithAttrl`. Of course, this processing is restricted to text nodes to ensure that only the text values of attributes and elements are converted:

```
convertToUtf :: (ArrowXml a) ⇒ a XmlTree XmlTree
convertToUtf
        = processTopDownWithAttrl editToUtf
            where
            editToUtf = changeText stringToUtf `when` isText
```

At last, this arrow has to be added to the list in `normaliseRDF` in order to use it during the normalisation process.

Typed Node Elements

It is common for RDF graphs to have `rdf:type` predicates from subject nodes. These are conventionally called *typed nodes* in a graph, or *typed node elements* in RDF/XML. They are used to describe resources as instances of specific types or classes. The next listing illustrates an example of typed node elements:

Listing 3.27: Typed Node Element

```
1 <?xml version="1.0"?>
2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3          xmlnx:exterms="http://www.example.org/terms/">
4   <rdf:Description rdf:about="http://www.example.org/staffid/85740">
5     <rdf:type rdf:resource="http://www.example.org/terms/person"/>
6     <exterms:name>John Smith</exterms:name>
7   </rdf:Description>
8 </rdf:RDF>
```

The example above does not contain an abbreviation and can be parsed by the `processRDF` arrow. But typed node elements appear more frequently in their shortened form, which has to be normalised before it is processed. The triple can be expressed more concisely by replacing the `rdf:Description` node element name with the namespaced-element of the value of the type relationship. The following listing shows how this can be achieved:

Listing 3.28: Concise Typed Node Element

```
1 <?xml version="1.0"?>
2 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3          xmlnx:exterms="http://www.example.org/terms/">
4   <exterms:person rdf:about="http://www.example.org/staffid/85740">
5     <exterms:name>John Smith</exterms:name>
6   </exterms:person>
7 </rdf:RDF>
```

The result of the normalisation of this example has to be similar to listing 3.27, which means that an additional property element with the name `rdf:type` has to be generated. Additionally, the node element name has to be replaced by `rdf:Description` while the original one has to be stored as the value of the property element. The arrow presented in the next listing performs exactly the essential steps. It uses the special predicate `isTypedNodeElem`, which holds, if an element is a node element without the name `rdf:Description`. Furthermore, the condition that the element is not empty has to be fulfilled, since the predicate `isTypedNodeElem` would also be true at property elements with the `rdf:nodeID` attribute:

Listing 3.29: Normalise Typed Node Elements

```
processTypedElem ::  (ArrowXml a) ⇒ a XmlTree XmlTree
processTypedElem
  = processTopDown (
      (insertChildrenAt 0 typeElem
       >>> setElemName rdf_Description)
      `when`
      (neg isEmptyElem >>> isTypedNodeElem))

    where
    typeElem
      = mkElement rdf_type
                  (qattr rdf_resource (getUniversalUri >>> mkText))
                  none

    isTypedNodeElem
      = isElem  >>> (hasQAttr rdf_nodeID `orElse` hasQAttr rdf_about)
                >>> neg (hasQName rdf_Description)
```

The arrow `insertChildrenAt` makes it possible to control the positioning of new elements in the list of child-nodes.

Property Attributes

When an object is a plain literal, it may be used as an attribute on the containing node element. Multiple properties on the same node element can be used as well, but only if the property element's name is not repeated, since attribute names have to be unique in an element. This abbreviation is known as *property attributes* and can be used on any node element. The next example illustrates the use of property attributes based on the document in listing 3.24:

```
1 <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
2                  dc:title="RDF/XML␣Syntax␣Specification␣(Revised)">
3   <exterms:editor>
4     <rdf:Description exterms:fullName="Dave␣Beckett">
5       <exterms:homePage rdf:resource="http://purl.org/net/dajobe/"/>
6     </rdf:Description>
7   </exterms:editor>
8 </rdf:Description>
```

The most obvious way of normalising this abbreviation is to generate property elements out of the property attributes. Every node element has to be tested for additional attributes besides the different RDF/XML specific ones. These additional attributes are those property attributes, out of which new property elements have to be created. The following listing shows the arrow that selects all non RDF/XML attributes and creates a new element out of them. This arrow has to be applied to every node element.

Listing 3.30: Create Elements out of Non-RDF/XML Attributes

```
propertyElements :: (ArrowXml a) ⇒ a XmlTree XmlTree
propertyElements
    = getAttrl
      >>> neg (hasQName rdf_about
               `orElse` hasQName rdf_nodeID)
      >>> arr mkqelem $<<< getQName
                          &&& listA none
                          &&& (listA (constA getChildren))
```

This listing illustrates the use of the special combinator (`$<<<`). Three values have to be calculated before they can be used to create the new property node. Both arrows, `getChildren` and `none` have to be handed over to `listA`, since these arrows have to be computed before they are added to the new element.

The implementation of the arrow, which applies `propertyElements` to every node element and inserts the new property elements into the list of child-nodes, is straightforward. The predicate `isNodeElem` does not follow an additional predicate that tests for non RDF/XML attributes. That is because `propertyElements` returns `none`, if there are no property attributes and then nothing happens which is the same as testing before applying:

Listing 3.31: Processing Property Attributes

```
processPropertyAttr ::(ArrowXml a) ⇒ a XmlTree XmlTree
processPropertyAttr
  = processTopDown (
      insertChildrenAt 0  propertyElements
      `when` isNodeElem )
```

With property attributes in node elements, it possible to define node elements without child-nodes. The described normalisation arrows would not be able to process the following node element:

```
<exterms:Book rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
              dc:title="RDF/XML Syntax Specification (Revised)"/>
```

This is a typed node element with a property attribute. It would not be detected and normalised by `processTypedElem`, since it is an empty node which has been excluded by the used predicates. Thus, the predicates of `processTypedElem` have to be extended so that empty elements with property attributes can be processed as well and the resulting elements would be as follows:

```
<rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
  <dc:title>"RDF/XML Syntax Specification (Revised)"</dc:title>
  <rdf:type rdf:resource="http://www.example.org/terms/Book"/>
</rdf:Description>
```

The final normalisation arrow contains more functions than before and is defined as follows:

Listing 3.32: Final Normalisation Arrow

```
normaliseRDF = processChildren $
                 seqA [convertToUtf
                      ,generateNodeID
                      ,processTypedElem
                      ,processChildren (multi isNodeElem)
                      ,deleteDuplicate
                      ,processPropertyAttr
                      ]
```

In order to keep the `rdf:RDF` element with all the namespace declarations of the source document in the resulting tree, the list of normalisation arrows is applied with `process-`

`Children` to the tree.
The order of all normalisation arrows in the list is significant. Of course, it does not matter when the encoding arrow `convertToUtf` is executed, since it is not influenced by the structure of the document tree. But `processPropertyAttr`, for example, can only be applied to the document after all node elements have been processed.

As a first conclusion, it has become evident that it is fairly easy to build a program with the Haskell XML Toolbox. Bigger problems can be separated into smaller tasks and then be combined, like it has been done with the several normalisation arrows and the RDF/XML parser. The next section will illustrate how a simple query language for RDF can be implemented and combined with the RDF/XML parser.

3.2.5. Simple Query Language

The query language which will be included into the RDF/XML parser is based on SPARQL. Since just some basic search features should be supported, only a subset of the full language will be implemented. These features are those which have been described in section 3.1.4. The grammar of the query language is listed in appendix B and shall not be discussed in detail. The EBNF format is the same as that used in the XML 1.1 specification. The productions in the grammar are almost the same as those of SPARQL, reduced by a number of language abbreviations and features.

A language parser has to be implemented, in order to provide the support of a special language in a program. The RDF/XML parser described above is very different from the one which is needed for the query language, because the real parsing of RDF/XML (i.e. analysing the syntax, generating tokens and finally create a data structure out of it) is provided by the XML parser of the Haskell XML Toolbox. The RDF/XML parser, on the other hand, processes the language on a higher level and adds additional semantics to the XML document.
The query language parser, however, must provide all steps of processing a language. The most common way of implementing a parser is to use a parser generator rather than to develop all the functionalities by one's own hand. In Haskell, there is the monadic parser combinator library *Parsec* [Parsec], which offers several advantages to the classic parser generators, e.g. it is written in Haskell and can be introduced easily into existing Haskell applications. The idea of parser combinators is to write small parsers for parts of the language and combine them to the final parser of a language. This is a common approach in Haskell. The Haskell XML Toolbox is also based on this idea, it provides several combinators, i.e. arrows, which allow to build complex arrows out of simpler ones. Parsec is part of the GHC compiler and has become a standard in Haskell. Even the XML parser of the Haskell XML Toolbox is based on Parsec which is another rea-

son to use Parsec for the query language as well. How a parser is written with Parsec shall not be discussed here because it is not the topic of this document. Only the main entry point to the parser and the connection between the query language parser and the RDF/XML parser will be described in detail.
The output of the parser is the SPARQL query stored in a special data structure. The main parser function is `parseSPARQL`, which has a `Query` as output. The type signature of the function is as follows:

Listing 3.33: Main Parser Function

```
parseSPARQL :: Parser Query
```

A function for applying this parser to a SPARQL query string is provided by Parsec. It is called `parse` and takes three arguments: the parser, the name of the input and the input itself. The input name is only used for error messages and can be empty. The actual use of the `parse` function will be illustrated in the next listing when the RDF/XML parser and the query language parser will be combined.
A SPARQL query conceptually consists of two things, the variables used in the SELECT-clause and the triple patterns of the WHERE-clause. The structure of a triple pattern is the same as of a normal RDF triple but it can have a variable instead of an RDF term in every position.
The data type `Query` has three constructors: the first one is used to represent the actual result of the parsing process, the second one is a normalised query where all abbreviations of the triple patterns, like object lists or predicate-object lists, have been removed, and the third one represents the empty query in case of an error during the parsing process. The next listing shows the definition of `Query` with its two type constructors:

```
data Query = Query [Var] GraphPattern
           | QueryN [Var] [TriplePattern]
           | Empty
           deriving (Show,Ord,Eq)

type Var = String
```

The first constructor uses a data type `GraphPattern` which contains differently abbreviated triple patterns. It will not be listed here, because only the constructor `QueryN` is of interest. This constructor represents the final result of the SPARQL parser and is used to perform the actual searching in the list of triples delivered by the RDF/XML parser. The type `Var` is a character string representing the name of a variable. A `TriplePattern` is defined as follows:

```
data TriplePattern = TriplePat Subject [(Predicate, [Object])]
                   | TriplePatN Subject Predicate Object
                   deriving (Show,Ord,Eq)
```

This data type has two variants, one is the abbreviated version which is used in `GraphPattern` and the other one is the normalised variant. It uses the data types `Subject`, `Predicate` and `Object`, which have already been defined and used in the implementation of the RDF/XML parser. In order to use these data types in the query language data structure, they have to be extended. The next listing shows the rewritten data types:

```
data Subject   = Subject RDFTerm
               | SubjectVar Var
               deriving (Ord,Eq)

data Predicate = Predicate URI
               | PredicateVar Var
               deriving (Ord,Eq)

data Object    = Object RDFTerm
               | ObjectVar Var
               deriving (Ord,Eq)
```

Now, every data type cannot only contain an RDF specific term, but also a variable. Of course, the variable constructor is not needed by the RDF/XML parser. The advantage of this approach is, that both parsers use the same data types which makes the parse results easier to combine, especially during the query evaluation. Like already mentioned, a subject may only be an URIref and not one of any other RDF terms that are represented by the data type `RDFTerm`. But still, the constructor `Subject` takes an `RDFTerm` and not an `URI`. That is because the specifications of RDF/XML and SPARQL contradict each other. In SPARQL, a subject can also be a literal, while this is still prohibited by RDF/XML. The RDF Core Working Group has noted, that the syntax may be extended to allow literals as the subjects of statements (see section 2.2 in [SPARQL]).

For every data type used in the SPARQL parser, there is a function which creates the normalised version of that data type. This normalisation is applied to the parser result before the query is evaluated. The type definition of the evaluation function is illustrated in the next listing:

Listing 3.34: Query Evaluation Function

```
evalQuery :: Query -> RDFStore -> [(Var,[(Either RDFTerm URI)])]
```

The function `evalQuery` takes a query generated by the SPARQL parser-function `parseSPARQL` and a list of triples. The result is a list of tuples where the first part is a variable of the SELECT-clause and the second part is a list of RDF values representing all possible bindings of the particular variable. These RDF values can either be a `RDFTerm` or an `URI`. The result can then be printed out in tabular form by the function `showQueryResult`.
The actual matching of the triple patterns to the triples is done by several functions which are used by `evalQuery`. Every triple pattern is compared with all triples and every possible binding is listed and returned. The list of bindings then contains any matching pairs of variables and RDF terms. This process is done in several steps, since it is quite a complex task. The functions which provide these functionalities shall not be listed here.
As a conclusion, two functions of the query language part are of interest. The first one is `parseSPARQL` which creates a Haskell data structure out of a SPARQL query string. The second one is the function `evalQuery` that performs the search defined in the SPARQL query on a list of triples. These triples can either be generated by another Haskell application or by the RDF/XML parser described above. How the result of the RDF/XML parser can be handed over to the query evaluation function will be presented in the next section.

3.2.6. Combining the SPARQL Parser and the RDF/XML Parser

The main function and the main processing function `processDocument` (see 3.12) have to be rewritten, in order to include the SPARQL parser into the RDF/XML parser. Furthermore, an additional commandline option should be added, which allow to choose between the real processing of the RDF/XML document and the normalisation without parsing. If the normalisation flag is set, the output of the parser will be RDF/XML and not the N-Triples format.
First, `processDocument` has to be extended by an additional parameter which is the SPARQL query character string and then it has three parameters: the commandline options stored in the `Attributes` type, the query string and the name of the input file:

```
processDocument :: Attributes -> String -> String -> IOSArrow b Int
```

The function `cmdLineOpts` now generates a triple with the list of attributes, the query string and the input file name. The new main function is shown in the next listing:

Listing 3.35: Final Main Function

```
main :: IO ()
main = do
       argv <- getArgs
       (al, qr, src) <- cmdlineOpts argv
       [rc] <- runX (processDocument al qr src)
       exitWith ( if rc >= c_err
                  then ExitFailure (-1)
                  else ExitSuccess )
```

The new commandline option is:

–*normaliseRDF* : document is normalised only

and has to be taken into consideration for two things: The option specify which processing arrows should be called and determine the output type of `writeDocument`, since it can be XML or plain text. Every commandline parameter is stored in the global state and can be therefore also accessed by the arrows dealing with the state. The following listing illustrates the predicate that tests if the option to normalise the RDF/XML document is set:

```
normaliseOption = getParamString a_normalise >>> isA (== "1")
```

The string `a_normalise` is the commandline option which can be set. Since the list of parameters is also accessible in `processDocument` directly, the lookup for the option `a_normalise` can also be done without the state arrows. With this predicate, the arrow `processDocument` can be rewritten:

Listing 3.36: Final processDocument

```
processDocument al qr src
  = readDocument al qr src
    >>>
    removeAllWhiteSpace >>> propagateNamespaces
    >>>
    ifA normaliseOption
        (normaliseRDF >>> indentDoc
            >>> writeDocument al outputFile)
        (replaceChildren (parseRDF >>> processQuery qr)
            >>> writeDocument ((a_output_xml,v_0):al) outputFile)
    >>>
    getErrStatus
```

If the normalise-option is set, the arrow `normaliseRDF` is applied to the tree, the resulting XML document is indented and the final result is passed to the arrow `writeDocument`. Else, if it is not set, the arrow `parseRDF` followed by `processQuery` is applied with `replaceChildren` to the tree and the result is also passed to `writeDocument`. The arrow `parseRDF` is just the normalisation followed by the RDF/XML processing arrow and the result of it is a `RDFStore`. The lookup for the output file parameter is done by `outputFile`. Before describing `processQuery`, another arrow has to be explained first. This is `getSPARQLQuery` which runs the parsing process of a query string and returns the query in the data structure. It uses the `parse` function of Parsec that has been already mentioned but not yet illustrated by an example. If an error occurs during the parsing, it is printed out and an empty query is returned, which means, that the evaluation process of this query is terminated and nothing is brought back. The actual input of the arrow is ignored. The listing 3.37 illustrates the definition of this arrow.

Listing 3.37: Query Parser Arrow

```
getSPARQLQuery :: String -> IOSArrow a Query
getSPARQLQuery queryStr
  = case (parse parseSPARQL "" queryStr) of
    Left parseError
    -> (issueFatal ("Syntax␣error␣in␣SPARQL␣query␣"
                        ++ show queryStr ++ ":␣"
                        ++ show parseError)
            >>> constA (Empty))
    Right theQuery
        -> constA (theQuery)
```

Finally, `processQuery` takes the generated `RDFStore` and the query string as an extra parameter. Out of them, it creates a `Query` with the function defined above and starts the evaluation process with them. The result is formated into a character string representation and then turned into a XML text node. If the query string is empty the whole `RDFStore` is printed without parsing and evaluating a query:

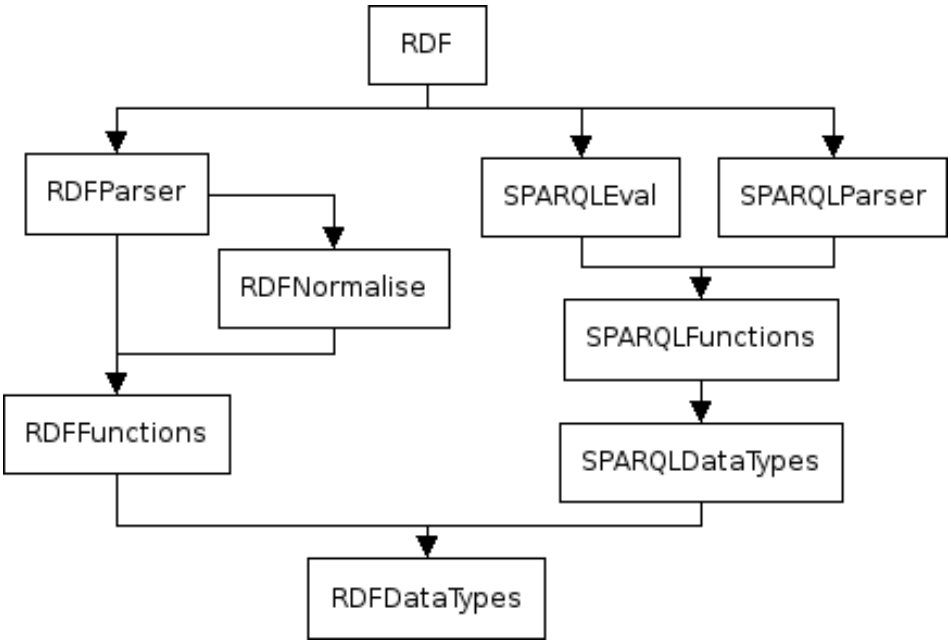
```
processQuery :: String -> IOSArrow RDFStore XmlTree
processQuery qr = ifP (const (qr == ""))
                      (arr showRDFStore)
                      ((getSPARQLQuery qr &&& this)
                        >>> arr2 evalQuery
                        >>> arr showResult)
                  >>> mkText
```

The result of `getSPARQLQuery` has to be combined with the identity arrow so that they build a tuple. The function `evalQuery` needs the result of the query-parse process and the input of the arrow, i.e. `RDFStore`, as parameters. The input of the arrow would be lost if the operator `(&&&)` with the identity arrow `this` would not have been used.

3.2.7. Module Hierarchy

The processing functions for RDF/XML and the functions of the SPARQL parser are structured in several modules. The module `RDF` is the main entry point and exports all the essential functions. `RDFDataTypes` defines all RDF specific data types while `SPARQLDataTypes` contains the type declarations for the query language parser. The module `RDFFunctions` includes the different predicates and the `QName`-definitions used during the RDF/XML processing.
The normalisation arrows are collected in `RDFNormalise`. This module provides various more arrows, additionaly to those which have been described. The arrows that parse the RDF/XML documents are defined in the module `RDFParser`. `SPARQLParser` contains the Parsec parsers for the query language grammar and `SPARQLEval` is the module which defines the evaluation functions of SPARQL queries. At last, the diverse functions for the query language, like function generating the character string representation, are in the module `SPARQLFunctions`. Figure 3.5 illustrates the structure and the dependences of these modules.

Figure 3.5: Module Hierarchy



4. Conclusion

4.1. Assessment of the Filter and Arrow Approach

It has already been mentioned, that the Haskell XML Toolbox has provided a different high-level programming interface than the discussed arrow approach. This interface is still available because of compatibility reasons. The idea of it is to use *filters* for manipulating XML documents. Different combinator functions give the possibility to create complex filters out of simple ones. The data structure used by these filters are the same as the one described above. The generic type `NTree` defines the tree structure and the specialised type `XmlTree` with `XNode` represents the XML document tree. The filter functions have one of the types shown in the next listing:

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

The first one defines a filter which takes a single generic tree and returns a list of trees, while the second filter takes a list of trees and also returns a list of trees. Of course, there are also specialised versions of these types, representing filters for XML trees:

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

The different filters can be separated in selector-functions, predicates, constructors and functions to manipulate nodes and subtrees, like it has been done with the different arrows. The boolean values **True** and **False** are represented by the none-empty list and the empty list, respectively, and there are also filters which embody the identity and the null case. Every function processing the XML document has to be of type `XmlFilter`, to ensure that they can be combined. One of the often used combinators is the operator `(>.)` which represents the sequential composition of filters. It is equivalent to the arrow composition with `(>>>)`.
For example, a filter which selects the text of a comment node has the following type:

```
getXCmt :: XmlFilter
```

One would expect, that a filter which selects the text of a comment node would return a character string. But that is not allowed, since the filters all have to be of the same type. The result of the filter `getXCmt` is therefore encapsulated by a text node stored in a list. If the processed node is not a comment node the resulting list would be empty. The type definition of the arrow providing the same functionality is shown in the next listing:

```
getCmt :: a XmlTree String
```

Comparing the arrow `getCmt` and the filter `getXCmt` exhibits the two disadvantages of the filter approach. The first problem is related to the programming style. As every filter function has to be of the same type, the type signature of it does not give any information about the way of operation of the filter. Only the name of the function and a possible description can help to discover what the filter actually does.
The second issue with the filter approach is also related to the type. The type checker of the Haskell compiler cannot detect any type errors. Haskell is a strong typed language, which is a big advantage. But by using the same type for all functions, the powerful type system of Haskell is practically switched off. The filter function `getXCmt` could also return a char reference or a new XML tag instead of the comment-text encapsulated by a text node. As a result, the type checker would not produce an error. This is because `XmlFilter` is a *type synonym*. There is no way to parameterise this type and to change the input and result type. The programmer has to rely on the description of the filter function, when he wants to process the output of a filter.
These problems become irrelevant when using arrows rather then filters. Arrows always have a specified type, the input and output type are specific and not always the same, since the instances of the arrow classes are data types defined with *newtype*. This makes the programming with them much more natural than with filters, since the type system of Haskell is able to react on type errors. Furthermore, the example of the RDF/XML parser has shown, that it is also possible to use other data types for special purposes with the arrows of the Haskell XML Toolbox. The filter approach would not allow this and the data structure of the XML parser has to be used. Everything has to be encapsulated by a `XmlTree`, although it is senseless.
The idea of using filters as the programming interface has been adopted from the XML parser HaXML [HaXML]. The filters of HaXML are all of the following type:

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

A filter works for nodes of the type `Content`. A `Content` represents the document subset of XML which is only a small part of the whole XML document. In contrast to the data model of the Haskell XML Toolbox, HaXML uses not a generic one but a more data

centric approach. The whole XML document is represented by different algebraic data types and almost every production of the XML grammar is modelled with a special type. Therefore, the filters cannot process the whole XML document and if one wants to work with other parts of the document, like the DTD, special functions have to be implemented.
The filters of the Haskell XML Toolbox instead give the possibility to process the whole generic data structure, hence the whole document. But nevertheless, HaXML has the same problems with the filter approach as the Haskell XML Toolbox. Type errors cannot be detected by the type system of Haskell.

4.2. Related Work

There are two other XML parsers written in Haskell, HaXML [HaXML] and HXML [HXML]. HaXML has already been introduced and since it has not an arrow module as the programming interface it will not be taken into consideration in this section. HXML, alternately, has changed its programming interface to arrows recently and will be compared with the Haskell XML Toolbox.
HXML is a non-validating parser and does not support XML namespaces but in return it is designed for space-efficiency. Moreover, HXML provides a special adapter to use it as a drop-in replacement for HaXML. The data model used in HXML is quite similar to the one in the Haskell XML Toolbox and has been a pattern for it. The structure of a XML document is modelled by the generic data type `Tree` and the document subset is represented as a `Tree` of `XmlNode`s:

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

DTDs are not stored in the tree model but in a special data type, in contrast to the Haskell XML Toolbox, where the DTD subset is also represented by the generic tree type. Hence, the approach of the Haskell XML Toolbox is much more general. This leads to the fact, that no extra processing functions for DTDs need to be implemented and the functions for processing the XML document subset and the DTD subset are the same.

HXML also uses arrows as the high-level programming interface. It follows the same idea as the Haskell XML Toolbox. The processing arrows represent computation over lists. In the Haskell XML Toolbox these arrows are called list arrows and in HXML they are named *filters*:

```
newtype Filter a b = Filter (a -> [b])
```

The arrow class which is implemented by `Filter` is provided by HXML and slightly different than the one used by the Haskell XML Toolbox. This is because the idea of arrows has been very new at the moment of the implementation and the compilers GHC or Hugs [Hugs] have not provided this class at that time. Therefore, the arrow class contains several combinators which are not included in the standard arrow class. There are no special combinators for list arrows like they are provided by the class `ArrowList`. This makes the programming interface of HXML less powerful then the one of the Haskell XML Toolbox. Especially, combinators which solve the problem of the point-free programming are missing. Furthermore, HXML does not provide any state handling. All in all, HXML seems to be in an experimental state in comparison to the Haskell XML Toolbox which offers more functionallity to serve as a professional XML parser.

4.3. Conclusion and Future Work

The examples of processing RDF/XML documents have shown that it is straightforward to implement applications with the programming interface of the Haskell XML Toolbox. The resulting RDF/XML parser and the query language extension are very short and compact programs. This is because of the functional language Haskell, which allows to develop in a very problem-oriented way.

The new approach of using arrows to process the XML document, has proven to be a flexible and useful way in comparison to the filter approach. Not only XML parsers, but also several other libraries for Haskell and other functional programming languages have adopted this approach recently. The idea of defining ones own notion of computations based on the specific problem provides an attractive programming style. The problem of arrows, using values more than once, i.e. the point-wise programming, can be solved with the arrow notation of Ross Patterson [Paterson 2001]. The special operators, provided by the Haskell XML Toolbox, are another way to deal with the problems of the point-free programming.

The intention of developing an RDF/XML parser and the SPARQL query language

parser was not to design a professional program. Instead, it has demonstrated the way of programming with the Haskell XML Toolbox. Nevertheless, the program parts can be extended easily in order to be used in a professional application, since the arrow interface and the functional approach have made the code understandable and maintainable. The different normalisation steps of RDF/XML can be improved so that they support all abbreviations of the RDF/XML syntax.

The Haskell XML Toolbox will be extended and maintained as well. At this moment, a parser for the schema language RelaxNG is written in the context of another master thesis. It also uses the arrow interface of the Haskell XML Toolbox.

Bibliography

[Bird 1988] *Introduction to Functional Programming using Haskell, second edition*, Richard Bird (1988), Prentice Hall Series in Computer Science, ISBN 0-13-48436-0

[CHARMOD] *Character Model for the World Wide Web 1.0*, Dürst M., Yergeau F., Ishida R., Wolf M., Freytag A., Texin T. (Editors), W3C Working Draft, 20 February 2002. This version is http://www.w3.org/TR/2002/WD-charmod-20020220/. The latest version is http://www.w3.org/TR/charmod/.

[DC] *Dublin Core Metadata Element Set, Version 1.1: Reference Description*, 02 June 2003. This version is http://dublincore.org/documents/2003/06/02/dces/. The latest version is http://dublincore.org/documents/dces/.

[GHC] *The Glasgow Haskell Compiler*, http://www.haskell.org/ghc

[HaXML] *HaXML: Haskell and XML*, Mallcom Wallace, http://www.cs.york.ac.uk/fp/HaXml/ .

[Hughes 2000] *Generalising monads to arrows*, John Hughes (2000), Science of Computer Programming, Volume 37.

[Hughes 2004] *Programming with Arrows*, John Hughes (2004), In AFP, Tartu, Estonia.

[Hugs] *Hugs 98* http://www.haskell.org/hugs/

[HXML] *HXML*, Joe Englisch, http://www.flightlab.com/ joe/hxml/ .

[Jones et al. 1998] *The Haskell 98 Report*, Simon Peyton Jones, John Hughes et al. (1998), http://www.haskell.org/onlinereport/ .

[N3] *Notation 3*, Tim Berners-Lee, http://www.w3.org/DesignIssues/Notation3 .

[Parsec] *Parsec: a free monadic parser combinator library for Haskell*, Daan Leijen, http://www.cs.uu.nl/∼daan/parsec.html .

[Paterson 2001] *A new notation for arrows*, Ross Paterson (2001), In ICFP, Firenze, Italy. ACM.

[Paterson 2003] *Arrows and computation*, Ross Paterson (2003), In Jeremy Gibbons and Oege De Moor, editors, The Fun of Programming. Palgrave.

[RDF Concepts] *Resource Description Framework (RDF): Concepts and Abstract Syntax*, Klyne G., Carroll J. (Editors), W3C Recommendation, 10 February 2004. This version is http://www.w3.org/TR/2004/REC-rdf-primer-20040210/. The latest version is http://www.w3.org/TR/rdf-concepts/.

[RDF Primer] *RDF Primer*, Manola F., Miller E., Editors, W3C Recommendation, 10 February 2004. This version is http://www.w3.org/TR/2004/REC-rdf-primer-20040210/. The latest version is at http://www.w3.org/TR/rdf-primer/.

[RDF Schema] *RDF Vocabulary Description Language 1.0: RDF Schema*, Brickley D., Guha R.V. (Editors), W3C Recommendation, 10 February 2004. This version is http://www.w3.org/TR/2004/REC-rdf-schema-20040210/. The latest version is http://www.w3.org/TR/rdf-schema/.

[RDF Sematics] *RDF Semantics*, Hayes P. (Editor), W3C Recommendation, 10 February 2004. This version is http://www.w3.org/TR/2004/REC-rdf-mt-20040210/. The latest version is http://www.w3.org/TR/rdf-mt/.

[RDF Tests] *RDF Test Cases*, Grant J., Beckett D. (Editors), W3C Recommendation, 10 February 2004. This version is http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/. The latest version is http://www.w3.org/TR/rdf-testcases/.

[RDF/XML Syntax] *RDF/XML Syntax Specification (Revised)*, Beckett D. (Editor), W3C Recommendation, 10 February 2004. This version http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/. The latest version is http://www.w3.org/TR/rdf-syntax-grammar/.

[Schmidt 2002] *Design and Implementation of a validating XML parser in Haskell*, Martin Schmidt (1999), Master's Thesis .

[SPARQL] *SPARQL Query Language for RDF*, Prud'hommeaux E., Seaborne A. (Editors), W3C Working Draft, 19 April 2005. This version is http://www.w3.org/TR/2005/WD-rdf-sparql-query-20050419/. The latest version is http://www.w3.org/TR/rdf-sparql-query/.

[SPARQL Result] *SPARQL Variable Binding Results XML Format*, Beckett D. (Editor), W3C Working Draft, 27 May 2005. This version is http://www.w3.org/TR/2004/WD-rdf-sparql-XMLres-20050527/. The latest version is http://www.w3.org/TR/rdf-sparql-XMLres/.

[URIS] *RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax*, Berners-Lee T., Fielding R., Masinter L., IETF, August 1998, http://www.isi.edu/in-notes/rfc2396.txt.

[XML] *Extensible Markup Language (XML) 1.0*, Second Edition, Bray T., Paoli J., Sperberg-McQueen C.M., Maler E. (Editors), World Wide Web Consortium, 6 October 2000. This version is http://www.w3.org/TR/2000/REC-xml-20001006. The latest version is http://www.w3.org/TR/REC-xml.

[XML-NS] *Namespaces in XML*, Bray T., Hollander D., Layman A. (Editors), World Wide Web Consortium, 14 January 1999. This version is http://www.w3.org/TR/1999/REC-xml-names-19990114/. The latest version is http://www.w3.org/TR/REC-xml-names/.

[XML Schema] *XML Schema Part 2: Datatypes*, Biron P., Malhotra A. (Editors), World Wide Web Consortium. 2 May 2001. This version is http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/. The latest version is http://www.w3.org/TR/xmlschema-2/.

A. List of Options for `readDocument` and `writeDocument`

Options for readDocument	
Option	**Description**
a_parse_html	use HTML parser, else use XML parser (default)
a_parse_html	use HTML parser, else use XML parser (default)
a_validate	validate document, else skip validation (default)
a_check_namespaces	check namespaces, else skip namespace processing (default)
a_canonicalize	canonicalise document (default), else skip canonicalisation
a_preserve_comment	preserve comments during canonicalisation, else remove comments (default)
a_remove_whitespace	remove all whitespace, used for document indentation, else skip this step (default)
a_indent	indent document by inserting whitespace, else skip this step (default)
a_issue_warnings	issue warnings, when parsing HTML (default), else ignore HTML parser warnings
a_issue_errors	issue all error messages on stderr (default), or ignore all error messages (default)
a_trace	trace level: values: 0 - 4
a_proxy	proxy for http access, e.g. www-cache:3128
a_use_curl	for http access via external programm curl, default is native HTTP access
a_options_curl	more options for external program curl
a_encoding	default document encoding (utf8, isoLatin1, usAscii, ...)

Options for writeDocument	
a_indent	indent document for readability, (default: no indentation)
a_remove_whitespace	remove all redundant whitespace for shorten text (default: no removal)
a_output_encoding	encoding of document, default is a_encoding or utf8

`a_output_xml`	(default) issue XML: quote special XML chars >,<,",',& add XML processing instruction and encode document with respect to a_output_encoding, if explicitly switched of, the plain text is issued, this is useful for non XML output, e.g. generated Haskell code, LaTex, Java, ...
`a_show_tree`	show tree representation of document (for debugging)
`a_show_haskell`	show Haskell representaion of document (for debugging)

B. Grammar of the Query Language

```
[1]  Query                      ::= SelectClause WhereClause
[2]  SelectClause               ::= 'SELECT' Var+ | 'SELECT' '*'
[3]  WhereClause                ::= 'WHERE' GraphPattern
[4]  GraphPattern               ::= '{' PatternElementsList '}'
[5]  PatternElementsList        ::= PatternElement PatternElementsListTail ?
[6]  PatternElementsListTail    ::= '.' PatternElementsList ?
[7]  PatternElement             ::= Triples
                                  | GraphPattern
[8]  Triples                    ::= VarOrTerm PropertyListNotEmpty
[9]  PropertyList               ::= PropertyListNotEmpty ?
[10] PropertyListNotEmpty       ::= Verb ObjectList PropertyListTail ?
[11] PropertyListTail           ::= ';' PropertyList ?
[12] ObjectList                 ::= Object ObjectTail ?
[13] ObjectTail                 ::= ',' ObjectList ?
[14] Verb                       ::= VarOrURI | 'a'
[15] Object                     ::= VarOrTerm
[16] VarOrURI                   ::= Var | URI
[17] VarOrTerm                  ::= Var | GraphTerm
[18] Var                        ::= <VAR>
[19] GraphTerm                  ::= RDFTerm
[20] RDFTerm                    ::= URI
                                  | RDFLiteral
                                  | NumericLiteral
                                  | BooleanLiteral
                                  | BlankNode
[21] NumericLiteral             ::= Integer | FloatingPoint
[22] RDFLiteral                 ::= String (<LANGTAG> | '^^' URI )?
[23] BooleanLiteral             ::= 'TRUE' | 'FALSE'
[24] String                     ::= <STRING_LITERAL1> | <STRING_LITERAL2>
[25] URI                        ::= QuotedURIref
[26] BlankNode                  ::= <BNODE_LABEL>
[27] QuotedURIref               ::= <Q_URIref>
[28] Integer                    ::= <INTEGER_10>
```

```
[29] FloatingPoint              ::= <FLOATING_POINT>
[30] <Q_URIref>                 ::= '<' ([^> ])* '>'  /* RFC 3869 */
[31] <BNODE_LABEL>              ::= '_:' (<NCNAME2>|<NCNAME1>)
[32] <VAR>                      ::= ('?'|'$') (<NCNAME2>|<NCNAME1>)
[33] <LANGTAG>                  ::= <AT> <A2Z>+ ('-' (<A2ZN>)+)*
[34] <AT>                       ::= '@'
[35] <A2Z>                      ::= [a-zA-Z]
[36] <A2ZN>                     ::= [a-zA-Z0-9]
[37] <INTEGER_10>               ::= <DIGITS>
[38] <FLOATING_POINT>           ::= [0-9]+ '.' [0-9]* <EXPONENT>?
                                    | '.' ([0-9])+ <EXPONENT>?
                                    | ([0-9])+ <EXPONENT>
[39] <EXPONENT>                 ::= [eE] [+-]? [0-9]+
[40] <STRING_LITERAL1>          ::= "'" ( ([^'\\\n\r]) | ('\\' [^\n\r]) )* "'"
[41] <STRING_LITERAL2|          ::= '"' ( ([^'\\\n\r]) | ('\\' [^\n\r]) )* '"'
[42] <DIGITS>                   ::= [0-9]+
[43] <NCCHAR1>                  ::= [A-Z]
                                    | [a-z]
                                    | [#x00C0-#x00D6]
                                    | [#x00D8-#x00F6]
                                    | [#x00F8-#x02FF]
                                    | [#x0370-#x037D]
                                    | [#x037F-#x1FFF]
                                    | [#x200C-#x200D]
                                    | [#x2070-#x218F]
                                    | [#x2C00-#x2FEF]
                                    | [#x3001-#xD7FF]
                                    | [#xF900-#xFFFF]
[44] <NCCHAR_END>               ::= <NCCHAR1> | '_' | '-' | [0-9] | #x00B7
[45] <NCCHAR_FULL>              ::= <NCCHAR_END> | '.'
[46] <NCNAME1>                  ::= <NCCHAR1> (<NCCHAR_FULL>* <NCCHAR_END>)?
[47] <NCNAME2>                  ::= '_' (<NCCHAR_FULL>* <NCCHAR_END>)?
```

C. 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.

Wedel, January 6, 2007