Implementing an XSLT processor for the Haskell XML Toolbox

A master thesis at the Fachhochschule Wedel by Tim Walkenhorst

August 31th, 2006

Supervisor: Prof.Dr. Uwe Schmidt

Declaration

I have written this thesis independently, solely based on the literature and tools mentioned in the chapters and the appendix. This document – in the present or a similar form – has not and will not be submitted to any other institution apart from the Fachhochschule Wedel to receive an academic grade.

Tim Walkenhorst, Wedel, August 31th, 2006

Contents

Introduction

Motivation

In this work we will implement a processor for a meaningful subset of the XSLT 1.0 specification based on Uwe Schmidt's Haskell XML Toolbox. Although there have been previous efforts to implement such a processor in Haskell (one at the University of Amsterdam by Danny van Velzen), none of the three major Haskell XML libraries (HaXml, HXML or HXT) do yet include an XSLT implementation.

It could be argued that it is not strictly necessary to include an XSLT processor within a library like the Haskell XML Toolbox, as it is always possible to call an existing processor like Apache's XALAN with a *System.Cmd.system* or a *System.Cmd.rawSystem* command-line-call. However such an approach suffers from the following problems:

- It is not possible to perform transformations on the level of XML trees. The trees would have to be serialized to a file before any transformation and parsed from a file whenever the result of a transformation is needed. This would be particularly expensive if we wanted to perform many small XSLT transformations on a tree with small non-XSLT transformation written in Haskell in between.

- It is not possible to store a compiled stylesheet and apply it on different occasions with this approach. This would be a problem whenever the compilation of a stylesheet is expensive compared to a single application of it.

The emphasis on this work is on the Haskell-implementation of XSLT. We will assume a reasonable level of experience with Haskell and XML, but no deep expertise in any of the two fields. While we will introduce the features of the XSLT language before implementing them, it is advisable to consult a textbook or an online tutorial, if you want to learn how to use the XSLT language. It is also recommended to use one of the freely available XSLT processors while developing stylesheets, to have a *"second opinion"* and to ensure maximum portability of the developed stylesheets. We use XALAN for this purpose. It is available as a binary distribution for most common platforms.

This work would not have been possible without Torben Kuseler's excellent XPath implementation which provides us with a necessary foundation on which we can build our system. My thanks go to my parents for their support and understanding in the last 29 years of my life, to my girlfriend for her patience with me during the time of the master thesis and to Uwe Schmidt for introducing me to some beautiful fields of computer science.

What exactly is XSLT?

XSLT is a language for certain kinds of transformations of XML-trees. Although any computable transformation of an XML tree can be expressed in XSLT, it is only suitable for certain kinds of common transformations including straight-forward restructuring and reordering of XML trees, adding and removing of boilerplate-code and extracting a relevant subset from an XML tree.

Let us consider a simple example. Imagine we have an XML format for invoices in which each item of the invoice has a name and a value:

```
<invoice>
  <item name="Steam hammer HXT 6.1" value="145.75" />
  <item name="Electric screwdriver GHC 6.4" value="69.49" />
  <item name="Medium sized screws 1000 pack" value="15.98" />
</invoice>
```

We can now write a simple XSLT transformation to generate an html document containing a table with the names of the items in the left column and the values in the right column:

```
<html xsl:version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <body>
    <table>
       <xsl:for-each select="*/item">
         <tr>
           <td><xsl:value-of select="@name"/></td>
           <td><xsl:value-of select="@value"/></td>
         </tr>
       </xsl:for-each>
    </table>
  </body>
</html>
```

This example shows a charming property many XLST transformations share: The stylesheet is remarkably similar to the document we want to create. The largest part of the stylesheet consists of literal result elements. These are simple XML elements which are not part of the XSL(T) namespace. The XSLT processor will copy these elements to the result document. The xsl:version attribute on the document element is needed to indicate that this is an XSL transformation. The *xsl:for-each* instruction selects all elements matching the XPath pattern **/item* in document order and instantiates its content for each of the elements. The *xsl:value-of* instruction selects the name- or value- attribute of the current element, converts it to a string and adds it to the result tree.

We can consult the popular literature for examples of what cannot be achieved with XSLT in a straight-forward manner. In chapter two *Numbers and Math* of Mangano's *XSLT Cookbook* it is shown how a simple logarithm can be computed with almost three pages of standard XSLT. Enough evidence to conclude that XSLT is not useful for anything but the most trivial numerical computations.

XSLT is often classified as a functional programming language based on the fact that variables and result tree fragments cannot be changed once they are created. Apart from that XSLT does not share much similarity with any classical functional programming language. In particular, XSLT does not provide any support for higher order functions, which are the base of the lambda calculus and the main building blocks of all larger functional programs. Therefore we simply classify XSLT as a special purpose programming language for certain kinds of transformations of XML trees.

Basic ideas for the implementation

Side note: Occasional forward references in this and the following chapters are inevitable as there are often reasons to implement a feature in a specific fashion based on *(sometimes obscure) requirements for the implementation of more advanced features. Readers unfamiliar with the XSLT specification can simply ignore them on the first reading.*

Any XSLT processor consists of two major parts. The first is a stylesheet compiler which transforms a stylesheet XML tree to an internal representation. The second is a stylesheet interpreter which interprets the compiled stylesheet and performs the transformation of the input document with the rules of the compiled stylesheet. We will develop both parts in parallel and will work our way inside-out through the XSLT specification. The simple and basic features are done first and we will go from there to the more complicated features. This way we can see some results early on.

We will call the internal representation of a stylesheet **abstract syntax**. For most elements of the concrete XML Syntax of the stylesheet there is an equivalent in the abstract syntax. However there is no one to one mapping between the abstract and concrete syntax. In particular different elements of the concrete syntax might be transformed to the same elements of the abstract Syntax. For example on the level of the abstract syntax *xsl:if* is treated as a special form of *xsl:choose* and literal result elements are mapped to the same symbols in the abstract syntax as element creation from *xsl:element* and *xsl:attribute*. Some of these features are implement slightly more general on the level of the abstract syntax to allow this mapping. We can consider literal result elements as a **derived form** of *xsl:element* and *xsl:attribute*. Whenever we identify an element from the XSLT specification as a derived form of another element which we have already implemented we merely have to handle the compilation of that feature. The interpretation is already implemented in terms of the internal language. The internal language is a language without any derived forms. Good explanations of the difference between concrete and abstract syntax, of derived form and of the internal and external languages can be found in Benjamin C. Pierce's *Types and Programming language*, in particular page 53 *abstract and concrete syntax*.

The compilation of a stylesheet is done conceptually in one pass. That does not mean that we always compile the nodes of the XML tree in document order. This means, however, that we will not jump up and down between different hierarchy levels of the XML tree and that we will not revisit a node once it has been compiled.

Imported stylesheets are compiled completely separate of each other. There is no common environment on which imported stylesheets on different hierarchy levels can rely. The **include** mechanism is an inclusion on the level of XML trees.

We implement stylesheet compilation exclusively as a transformation from an XML tree to an internal representation. With a SAX like programming interface the stylesheet could be compiled directly from a file to the internal representation without the need for creating an intermediate XML tree. However, it seems that all current Haskell XML libraries exclusively implement tree based APIs. This could be an interesting topic for further research.

A gentle start - Basic instructions

Although the XSLT specification and most books on XSLT start with an explanation of the skeleton of a stylesheet, we will start with an explanation of the innermost structures of a stylesheet where *"the real work is done"*. Implementing these features in the beginning allows us to quickly build a working kernel and have some useful material at hand once we start to implement the more challenging features. This presentation is also consistent with the way compiler construction is usually presented in textbooks. It starts with expressions and continues with statements, variables, procedures and modules in that order. We basically start with statements, since expressions have been previously defined as the XPath language.

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <!-- the content of our simplified stylesheets -->
  </xsl:template>
</xsl:transform>
```

We simply consider the content of a template as a stylesheet. This *"stylesheet"* is evaluated with the root-node of the input document as the current context. We use a small script to add the top- and bottom- two lines to our simplified stylesheets and can then match our output against XALAN's output. We choose a *down-to-the-metal* (, or literal programming) approach in the presentation. Almost every feature will be explained with its exact Haskell code. The entire Haskell code is small enough to print out and is listed in appendix IV. This presentation is influenced by Niklaus Wirth's *Grundlagen und Techniken des Compilerbaus*. He manages to systematically implement a working compiler for a meaningful subset of Oberon within one rather short book. He does not make use of any advanced tools like parser generators and is still able to provide the source code for his compiler in less than thirty pages. We attempt the same here for XSLT.

Creating literal text

The *xsl:text* instruction is used to generate a text node in the result tree. We will represent any *xsl:text* instruction in the concrete XML syntax by a corresponding data constructor in Haskell:

```
data Template = TemplText String
```

The word template is used here for any kind of instruction, literal result element, text-node which must be copied to the result tree, etc. Basically anything that can be expanded to create nodes in the result-tree. It must not be confused with *xsl:template* which we will either call a named rule or a match rule (and sometimes both) depending on the context.

```
compileText :: XmlTree -> Template
compileText = TemplText . collectTextnodes . getChildren
```

Assuming we have already identified the current node in the result tree as *xsl:text* the function shown above will collect all text-nodes below that node and combine them to one string. For example *<xsl:text>some<!-- -->string</xsl:text>* is compiled to *TemplText "somestring"*. *(C)ollectTextnodes* is implemented as follows:

```
collectTextnodes :: [XmlTree] -> String
collectTextnodes = concat . mapMaybe getText
```

The Haskell XML toolbox (HXT) has provided us with the necessary primitive functions *getChildren* and *getText*. The former is actually a function which is not restricted to XML trees but can be used on any rose tree. In a rose tree each node can have arbitrary many children, or more formally each node has a forest of children. The function *getText* is one of several overloaded functions which will work either on a single node (of type *XNode*) or on an *XmlTree* (a rose tree of *XNode*[s]). In the second case the function is simply applied to the root node. It returns *just* the string value for a text node and *Nothing* for any other node.

The application of the text template is reasonably simple:

```
applyText :: Template -> Context -> [XmlTree]
applyText (TemplText s) _ = [mkText s]
```

(A)pplyText is one of the few template application functions that are totally independent of the context in which it is applied. Nonetheless we will consistently use the type *Template -> Context -> [XmlTree]* for any of these functions.

Literal text can be created in XSLT by either *xsl:text* elements or literal text in the stylesheet. They only differ in the way whitespace is treated. We will use the following compilation function for literal text nodes in a stylesheet:

```
compileTextnode :: XmlTree -> Template
compileTextnode = TemplText . fromJust . getText
```

The text template application is not affected by the new feature in any way. *(X)sl:text* and literal text templates are indistinguishable on the level of the abstract syntax. Whitespace handling will be discussed on page 59. The handling of the obscure and optional disable-output-escaping attribute is examined on page 70.

Empty and combined templates

Wherever a template like *xsl:text* may appear within a stylesheet it is also allowed to use more than one template or no template at all. We introduce a new data constructor to handle these situations:

```
data Template = ...
  | TemplComposite [Template]
```

An empty template is represented by *TemplComposite []*.

This brings us to the general structure of the necessary *compileTemplate* function:

```
compileTemplate :: [XmlTree] -> Template
compileTemplate [node]       =
   if isElem node
   then let elemName = fromJust $ getElemName node in
        if      equivQName elemName xsltText  then compileText node
        else if equivQName elemName xslt...   then compile...  node   -- each allowed xsl:* element

        -- no other xslt elements allowed here:
        else if namespaceUri elemName == xsltUri
        then error $ "xslt-element " ++ localPart elemName ++ " not allowed within this context."
        -- for now all other elements will be considered as literal result elements:
        else compileLiteralResultElement node

   else if isText node then compileTextnode node

   else
       error $ "Unsupported node type in xslt sheet: " ++ show (getNode node)
compileTemplate list = compileComposite list
```

Single element-nodes are compiled with their respective compilation functions, of which we have already demonstrated *compileText*. Unknown elements from the XSLT namespace will not be allowed. None XSLT elements will be interpreted as literal result elements (*p. 14*) the only allowed non-element types are literal text nodes which we have already discussed. Some care has to be taken to avoid an infinite mutual recursion between *compileTemplate* and *compileComposite*. The later can be defined as follows:

```
compileComposite :: [XmlTree] -> Template
compileComposite = TemplComposite . map (compileTemplate . return)
```

The application can be defined as follows:

```
applyTemplate :: Template -> Context -> [XmlTree]
applyTemplate t@(TemplComposite _)     = applyComposite t
applyTemplate t@(TemplText _)          = applyText t
applyTemplate t@(Templ... _)           = apply... t        -- for all TemplXY...
```

For now *applyComposite* can be implemented as below, but we will have to revisit it once we deal with XSLT variables.

```
applyComposite :: Template -> Context -> [XmlTree]
applyComposite (TemplComposite templates) ctx =
    concatMap (flip applyTemplate ctx) templates
```

After these first steps we can run our first simplified stylesheet transformations. For example the stylesheet *Som<xsl:text>est</xsl:text>ring* will generate the output *Somestring* for any input document. Hardly impressive, but we have a start.

How do we treat the input document?

While the previous stylesheet fragments simply ignored the input document and produced some constant results (, or result tree fragments in the language of the XSLT specification), we must keep track of some context. The minimal requirements for this context are described in the XSLT- and the XPath-specification.

Initially the context can be defined as follows:

```
data Context = Ctx NavXmlTree                -- current node
                   [NavXmlTree]              -- current node list
                   Int                       -- position of current node 1..length
                   Int                       -- length of node list
               | CtxEmpty
```

There is always a current node which is selected to be transformed. Initially the current node is the root node of the input document. One or more nodes can be selected as a current node list. Two integer attributes are used to store the position of the current node within the current node list and the length of the current node list. Initially the values of both are one. *CtxEmpty* can be used when some selection returns an empty set of nodes.

A careful reader might have noticed that the type *NavXmlTree* is used here instead of *XmlTree*. This results from an XPath requirement. A stylesheet tree can be compiled by just recursively descending into the tree. No further navigation is needed. A result tree (-fragment) of a transformation step does not allow any upward traversal as the ancestors of the each current result fragment do not yet exist. However, XPath does allow navigating arbitrarily within the input document. We can select the ancestor of the current node with ".." or even directly refer to the root node with "/". Therefore navigatable rose trees are needed. Most of the time, we simply pass these navigatable trees to Torben Kuseler's XPath implementation. However, on the few occasions we actually do something meaningful with these trees we will introduce the needed operations.

Computing text

The *xsl:text* and literal text elements do not refer to the input document in any way. Certainly any interesting stylesheet will not just create some constant result tree. It will extract and combine parts of the input document to create some interesting result document. It seems instructive to start with the simplest possible template which actually does extract something from the result tree: the *xsl:value-of* template. In our introductory XSLT example we have used

```
<xsl:value-of select="@name"/>
```

to extract the string-value of the *name* attribute from an element. In the abstract syntax value-of can be represented by:

```
data Template = ...
  | TemplValueOf StringExpr
```

where:

```
newtype StringExpr = StringExpr Expr
```

Expr is an XPATH expression. It is defined in the XPath module. *StringExpr* adds a type-information to an XPATH expression. It is used as documentation which can be enforced by the compiler. We use newtype instead of data here because it allows us to wrap and unwrap the XPath expression to and from a string expression without any runtime penalty. *(V)alue-of* is compiled as follows:

```
compileValueOf :: XmlTree -> Template
compileValueOf node =
    TemplValueOf $ parseStringExpr $ fetchAttribute node xsltSelect
```

According to the XSLT specification the result of the select expression is converted to a string as if by a call to the XPath string-function:

```
parseStringExpr :: String -> StringExpr
parseStringExpr = StringExpr . mkStringExpr . parseExpr

-- This is equivalent to:
-- parseStringExpr e = StringExpr $ parseExpr $ "string(" ++ e ++ ")"
```

(M)kStringExpr adds a call to the *string* function to a compiled XPath expression as if the literal XPath expression was surrounded in a call to the *string()* function. As shown, it could be implemented literally. However, since such string-expressions do occur frequently it is advisable to save the parsing time for the call to the *string* function.

The application of *value-of* is straight forward:

```
applyValueOf :: Template -> Context -> [XmlTree]
applyValueOf (TemplValueOf expr) ctx = [mkText $ applyStringExpr expr ctx]
```

where:

```
applyStringExpr :: Expr -> Context -> String
applyStringExpr (StringExpr expr) ctx = string
  where (XPVString string) = evalXPathExpr expr ctx
```

It's legitimate to force the *XPVString* pattern here, since a non-string result could only stem from implementation errors of the XPath or XSLT module, but not from an error in the stylesheet. The job of *evalXPathExpr* is to translate the XSLT context to the XPath context. We will see that the former contains values which are not needed and understood by the XPath module. For now the implementation is simple:

```
evalXPathExpr :: Expr -> Context -> XPathValue
evalXPathExpr expr (Ctx node _ pos len) =
    filterXPath $ evalExpr ([],[]) (pos, len, node) expr (XPVNode [node])
  where
    filterXPath (XPVError err) = error err
    filterXPath xpv            = xpv
```

(E)valExpr is a function from the XPath module. The first argument tuple ([], []) is an empty environment of variable- and key bindings. We will come back to that later. The *XPVNode [node]* argument seems redundant. There is a need to pass an *intermediate result* to XPath, as the *eval* function will sometimes be called internally by XPath to evaluate a sub-expression with an intermediate result. However, when calling XPath from the outside passing a node set consisting exclusively of the context node is the *<u>only</u>* valid choice. It took a while to figure this out.

Attribute value templates

Before we discuss how elements and attributes can be created, it is advisable to examine one common XSLT feature first: attribute value templates. XSLT provides a convenient notation for mixing literal text with computed text from an XPath expression. Attribute value templates occur throughout XSLT. For example the name and namespace attributes of *xsl:element* and *xsl:attribute* and the data-type and order attributes of *xsl:sort* are attribute value templates (AVT). AVTs consist of literal text in which XPath expressions can be embedded within a pair of curly braces. Literal curly braces are quoted by double curly braces, that is *{{* means *{* and *}}* means *}*. AVTs do not increase the expressive power of XSLT, as there is always an equivalent XPath expression. The table below shows a few examples.

<u>AVT:</u>	<u>equivalent XPath-expression:</u>
`""`	`"''"`
`"Hello"`	`"'Hello'"`
`"{.}"`	`"string(.)"`
`"Hello {1+1} you"`	`"concat('Hello ',1+1,' you')"`

```
"Hello {{{1+1}}} you"                           "concat('Hello {',1+1,'} you')"
```

In order to transform an attribute value template to an XPath expression we need to split the AVT into a sequence of tokens for literal text including quoted curly braces and embedded XPath expressions. It would be inefficient and troublesome to implement the construction of the XPath-expression on the level of the concrete syntax (i.e., by string twiddling). We extent Torben Kuselers XPath module with a few basic functions to build these expressions. This approach also relieves us from the subtleties of adding the correct type of " or ' characters around the literal expression as we see below:

```
mkLiteralExpr :: String -> Expr
mkLiteralExpr = LiteralExpr
```

The creation of a literal expression is just an alias for an XPath data constructor. However, we prefer to not access the XPath constructors directly from the XSLT module. Creating a *concat* expression is slightly more complicated:

```
concatExpr :: [Expr] -> Expr
concatExpr []                    = LiteralExpr ""
concatExpr [lit@(LiteralExpr _)] = lit
concatExpr xs1@[_]               = FctExpr "string" xs1
concatExpr xs                    = FctExpr "concat" xs
```

(C)oncatExpr implements the transformation, which is shown in right column of the table above, on the level of the abstract XPath syntax. The special casing for a single literal expression is strictly an optimization. Adding a call to the XPath *string* function wouldn't hurt here, but is unnecessary. All cases could be merged into one, if the *concat* function was specified more lenient and accepted an arbitrary number of arguments instead of two or more arguments. We cannot just sneak in an arbitrary number of arguments as the checking if a function is available and has the correct number of arguments must be deferred until runtime to allow the use of possible extension functions. If not supported, the use of such a possible extension function must only lead to an error if the expression is actually evaluated.

Based on the previous considerations we can implement the compilation of an attribute value template to an XPath expression by straight-forward recursion and pattern matching:

```
parseAVT :: String -> StringExpr
parseAVT str =
    StringExpr $ concatExpr $ splitAVT str ""
  where

    splitAVT :: String -> String -> [Expr]
    splitAVT ""           acc = acc2lit acc
    splitAVT ('{':'{':xs) acc = splitAVT xs $ '{':acc
    splitAVT ('}':'}':xs) acc = splitAVT xs $ '}':acc
    splitAVT ('{':xs)     acc = let (body, rest) = span (`notElem` "{}") xs in
                                if not (null rest) && head rest == '}'
                                  then acc2lit acc ++ parseExpr body : splitAVT (tail rest) ""
                                  else error $ "Unterminated expression " ++ xs ++ " in AVT."
    splitAVT ('}':_)      _   = error $ "deserted '}' in AVT."
    splitAVT (x:xs)       acc = splitAVT xs $ x:acc

    acc2lit :: String -> [Expr]
    acc2lit ""  = []
    acc2lit acc = [mkLiteralExpr $ reverse acc]
```

Someone might complain that this implementation is too low level and doesn't make use of the *appropriate tools* like regular expressions or even parser generators. Actually, we have implemented both alternative approaches and found that this implementation turned out to be the shortest and easiest to understand for the simple problem at hand. For those who want the comparison, a regular expression version and a version using the Parsec parser combinator library are listed in appendix III p.72 .

Creating elements and attributes

XSLT defines the following templates for the creation of elements and attributes:

```
<xsl:element name = { qname } namespace = { uri-reference } use-attribute-sets = qnames>
   <!-- content: template -->
</xsl:element>

<xsl:attribute name = { qname } namespace = { uri-reference }>
   <!-- content: template -->
</xsl:attribute>
```

If we ignore *use-attribute-sets* for the moment, both templates require a *name* attribute value templates (AVT) and allow an optional namespace AVT for namespace-qualified names. It seems reasonable to extract this commonality and define a dedicated data type for qualified names which are computed from AVTs:

```
data ComputedQName = LiteralQName QName
                   | CompQName StringExpr -- name
                               StringExpr -- namespace
```

The data constructor *LiteralQName* will become handy once we deal with literal result elements. Based on this type we can express the data-constructors for elements and attributes as follows:

```
data Template = ...
  | TemplElement ComputedQName Template
  | TemplAttribute ComputedQName Template
```

For now *compileElement* is defined as follows:

```
compileElement :: XmlTree -> Template
compileElement node =
    TemplElement (compileComputedQName node) $ compileTemplate (getChildren node)
```

where:

```
compileComputedQName :: XmlTree -> ComputedQName
compileComputedQName node =
    CompQName nameAVT nsAVT
  where
    nameAVT  = parseAVT $ fetchAttribute node xsltName
    nsAVT    = parseAVT $ fetchAttributeWDefault node xsltNamespace ""
```

The namespace attribute has the empty string as a default. *(C)ompileAttribute* is defined like *compileElement*.

For now, the application of a computed name and an *xsl:attribute* are reasonably simple:

```
applyComputedQName :: ComputedQName -> Context -> QName
applyComputedQName (LiteralQName qName) ctx = qName
applyComputedQName (CompQName nameExpr nsExpr) ctx =
    mkNsName (applyStringExpr nameExpr ctx) (applyStringExpr nsExpr ctx)

applyAttribute :: Template -> Context -> [XmlTree]
applyAttribute (TemplAttribute compQName template) ctx =
    [mkAttr qName content]
  where
    qName   = applyComputedQName compQName ctx
    content = applyTemplate template ctx
```

However, we have to be a bit more careful with the application of an element. Even though it does not seem reasonable to add many attributes with the same name to an element, this situation can happen once we deal with literal result elements or attribute sets. To cope with it we will ensure two things: The attributes with the highest priority will be added last. And: When constructing the attribute list, of each class of attributes with identical names only the one which has been added last to the stylesheet will be used. The following implementation ensures the second requirement:

```
applyElement :: Template -> Context -> [XmlTree]
applyElement (TemplElement compQName template) ctx =
    return $ createElement qName content
  where
    qName   = applyComputedQName compQName ctx
    content = applyTemplate template ctx
```

where:

```
createElement :: QName -> [XmlTree] -> XmlTree
createElement name fullcontent =
    mkElemen name distinctAttribs content
  where
    distinctAttribs    = nubBy eqAttr $ reverse attribs
    (attribs, content) = span (isAttr) fullcontent
    eqAttr node1 node2 = equivQName (fromJust $ getAttrName node1) (fromJust $ getAttrName node2)
```

(C)reateElement splits the created result tree fragment into the attribute list and the content of the element using span. *(N)ubBy* and *reverse* ensure that in the case of a name collision only the attribute which was added last is used.

Literal result elements

The names of elements and attributes are not always computed. Whenever the name of an element which should be added to the result tree is known statically, the element and all of its attributes with statically known names can be added via a convenient syntax: Literal result elements (LRE). LREs have already been used in our initial example to create html elements.

Whenever an element is neither an XSLT template nor an extension element it will be interpreted as a literal result element by the XSLT processor.

Any LRE can be expressed in terms of an an equivalent sequence of *xsl:element*, *xsl:attribute* and *xsl:value-of* as the following mapping illustrates:

```
                                                          <xsl:element name="lre">
                                                            <xsl:attribute name="attr_1">
                                                              <xsl:value-of select="xp_1"/>
<lre attr_1="avt_1" ... attr_k="avt_k">                      </xsl:attribute>
  content                                         ≝          ...
</lre>                                                      <xsl:attribute name="attr_k">
                                                              <xsl:value-of select="xp_k"/>
                                                            </xsl:attribute>
                                                            content
```

```
                                                        </xsl:element>
```

<u>*where:*</u> xp_i *is an XPath expression which is equivalent to* avt_i.

Since attribute value templates are already compiled to equivalent XPath expressions, the translation from avt_i to xp_i is trivial. As already shown *ComputedQName* has an additional data constructor *LiteralQName* which comes handy when we want to create element and attribute templates from literal result elements.

```
compileLiteralResultElement :: XmlTree -> Template
compileLiteralResultElement node =
    TemplElement compQName content
  where
    compQName         = LiteralQName $ fromJust $ getElemName node
    content           = TemplComposite $ attributes ++ [template]
    attributes        = mapMaybe compileLREAttribute $ fromJust $ getAttrl node
    template          = compileTemplate (getChildren node)
```

For all attributes of a LRE which are neither namespace-declarations nor a part of the XSLT namespace an attribute template is created.

```
compileLREAttribute :: XmlTree -> Maybe Template
compileLREAttribute node =
    if isSpecial
      then Nothing
      else Just $ TemplAttribute (LiteralQName name) val
  where
    isSpecial = namespaceUri name `elem` [xsltUri, xmlnsNamespace]
    name      = fromJust $ getAttrName node
    val       = TemplValueOf $ parseATV $ collectTextnodes $ getChildren node
```

In her master thesis *Konzeption und Design eines XSLT Prozessors unter dem Aspekt der funktionalen Programmierung in Haskell* Christine Apfel suggests that some parts of a stylesheet and LREs in particular might be subject to constant folding. Within our code we could introduce a new data constructor *TemplConst [XmlTree]* and propagate the constant values upwards with compilation process. However, these constant tree fragments would still be subject to namespace aliasing, which couldn't easily be performed within this compilation step. It is also not clear whether great performance gains could be achieved with this sort of constant folding, but it could be an interesting thing to play around with when it comes to optimizing the XSLT processor.

Namespace aliasing and attribute sets will be discussed on page 61 and page 55, respectively.

Conditional processing

XSLT provides the multi-branch conditional *xsl:choose*:

```
<xsl:choose>
  <!-- content: xsl:when+, xsl:otherwise? -->
</xsl:choose>

<xsl:when test = boolean-expression>
  <!-- content: template -->
</xsl:when>

<xsl:otherwise>
  <!-- content: template -->
</xsl:otherwise>
```

(X)sl:choose instantiates the content of the first *when* part in document order for which the value of the *test* expression converted to a boolean returns *true()*. If no *when* part applies and there is an *otherwise* part, the content of the *otherwise* part is instantiated. If no *when* part applies and there is no *otherwise* part an empty result tree fragment is returned.

(X)sl:otherwise can be regarded as a derived form of xsl:when:

```
<xsl:otherwise>                          <xsl:when test="true()">
  content                   ≝              content
</xsl:otherwise>                         </xsl:when>
```

The abstract syntax is defined as follows:

```
data Template = ...
  | TemplChoose [When]
```

where:

```
data When = WhenPart TestExpr Template
```

Similar to the previously discussed string expressions, test expression are wrapped within a newtype declaration to enable some type-checking at compile-time:

```
newtype TestExpr = TestExpr Expr
```

The compilation of an *xsl:choose* template is shown below. First all non-element nodes will be stripped from the content. This is necessary as there might be whitespace nodes which are selected to be preserved by *xml:space*, but must be ignored in this situation. The implementation is somewhat more lenient than required by the XSLT specification, as it does allow zero *when* parts or an *otherwise* part followed by one or more *when*- or *otherwise* parts. It's unlikely that this causes any problems in practice, since these are not very common programming errors.

```
compileChoose :: XmlTree -> Template
compileChoose node = TemplChoose whenParts
  where whenParts  = map compl children
        children   = filter isElem (getChildren node)
        compl node' = let elemName = fromJust $ getElemName node' in
                        if      equivQName elemName xsltWhen      then compileWhen node'
                        else if equivQName elemName xsltOtherwise then compileOtherwise node'
                        else error $ show elemName ++ " not allowed within xsl-choose template!"
```

where:

```
compileWhen :: XmlTree -> When
compileWhen node = WhenPart expr $ compileTemplate $ getChildren node
  where expr     = parseTest $ fetchAttribute node xsltTest
```

The XSLT specification states that the result of the *xsl:test* expression is converted to a boolean as if by a call to the *boolean* function.

```
parseTest :: String -> TestExpr
parseTest = TestExpr . mkBoolExpr . parseExpr

-- This is equivalent to:
-- parseTest s = TestExpr $ parseExpr $ "boolean(" ++ s ++ ")"
```

Basically the same discussion we had with *parseStringExpr* on page 10 can be applied here. While the literal implementation is possible, the chosen implementation is more efficient as no parsing needs to be done to add the call to the boolean function to the compiled expression.

The translation from *xsl:otherwise* to *xsl:when test="true()"* is implemented almost literally below:

```
compileOtherwise :: XmlTree -> When
compileOtherwise node = WhenPart (TestExpr mkTrueExpr) $ compileTemplate $ getChildren node
```

The application of *TemplChoose* is implemented by straight forward recursion over the *when* list:

```
applyChoose :: Template -> Context -> [XmlTree]
applyChoose (TemplChoose whenList) ctx = applyWhenList whenList ctx
```

where:

```
applyWhenList :: [When] -> Context -> [XmlTree]
applyWhenList []  _ = []
applyWhenList ((WhenPart expr template):xs) ctx =
  if applyTest expr ctx
    then applyTemplate template ctx
    else applyWhenList xs ctx
```

(A)pplyTest can be defined similar to *applyStringExpr*. Again, forcing the *XPVBool* pattern is justified here, since a return value of a different type would indicate a programming error in the XSLT or the XPath module and not an error in the stylesheet.

```
applyTest :: TestExpr -> Context -> Bool
applyTest (TestExpr expr) ctx = bool
  where (XPVBool bool) = evalXPathExpr expr ctx
```

XSLT provides a single-branch if, which can be treated as a derived form of *xsl:choose*:

```
                                              <xsl:choose>
<xsl:if test=expr>                              <xsl:when test=expr>
  content                       ≝                 content
</xsl:if>                                       </xsl:when>
                                              <xsl:when>
```

The compilation of *xsl:if* is implemented below:

```
compileIf :: XmlTree -> Template
compileIf = TemplChoose . return . compileWhen
```

Repetition

None of the templates we discussed so far changes the context in any way. Initially the current nodes consist only of the root node of the input document. If we want to transform an arbitrary number of similar nodes with one template we need to able to select all these nodes with an expression and process them in some order (, usually in document order). More technically one template is instantiated for each node of a selection, whereby each node of that selection is used once as the current context node.

In XSLT we have:

```
<xsl:for-each select = node-set-expression >
  <!-- content: xsl:sort*, template -->
<xsl:for-each>
```

Sorting will be explained on page 52 ff. Ignoring sorting for the moment the abstract syntax of a for-each template can be described with the following data constructor:

```
template = ...
  | TemplForEach SelectExpr Template
```

The new expression type *SelectExpr* differs from the previously described expression types in the result-type which must be a *node-set*. Regardless of their name, node sets have an individual ordering. Therefore node-sets should really be called *node lists*. Let us focus on the select expressions for the moment.

The compilation of a select-expression is defined as:

```
parseSelect :: String -> SelectExpr
parseSelect = SelectExpr . parseExpr
```

While the application is defined as:

```
applySelect :: SelectExpr -> Context -> [NavXmlTree]
applySelect (SelectExpr expr) ctx =
    extractNodes xpathResult
  where
    xpathResult                  = evalXPathExpr expr ctx
    extractNodes (XPVNode nodes) = nodes
    extractNodes r               = error $ "wrong type ..."
```

Select expressions are inferior to the previously defined expressions. Test- and string expression would always return a boolean or string. This could be achieved by a call to the string- or the boolean function. Unfortunately there is no nodeset function. It is simply not possible to convert an arbitrary XPath value to a nodeset in a sensible way. Imagine we converted non-nodeset values to text nodes. Then what would be the ancestor- or root-nodes of these text nodes? Of course we could convert all non-nodeset values to either the empty nodeset or a nodeset with an empty root node, but this doesn't sound like a useful feature either. Because of this or other reasons using an expression which does not return a nodeset is an error. Whenever this situation occurs an error is signaled when the respective select expression is evaluated. The next interesting questions is whether it would be possible to detect type errors in select expressions statically, that is when the expression is *compiled*. The presence of variable expressions makes this extremely hard, since the type of variables and parameters is a runtime property. It would be possible to perform some sanity checking. For example literal strings or numbers and calls to string- or number-functions could be disallowed. However, the XSLT specification is not clear if such a sanity checking would even be allowed, since the illegal select-expression might never be reached. Therefore we restrict our implementation to runtime type-checking alone.

We leave the *compileForEach* function for the simple *for-each* template without sorting as an exercise for the inclined reader. The application is defined as follows:

```
applyForEach :: Template -> Context -> [XmlTree]
applyForEach (TemplForEach expr template) ctx =
    processContext (ctxSetNodes (applySelect expr ctx) ctx) $ applyTemplate template
```

It makes sense to provide some operations on the context at this point. *(C)txSetNodes* creates a new context from nodeset and sets the the current node to the first node of the nodeset. For an empty nodeset *CtxEmpty* is returned. Additional attributes of the old context will be copied to the new context. These additional attributes will be added to the context in the following chapters whenever they are needed.

```
ctxSetNodes :: [NavXmlTree] -> Context -> Context
ctxSetNodes _     CtxEmpty = error "ctxSetNodes: Internal error attempt to access the empty context"
ctxSetNodes []    _        = CtxEmpty
ctxSetNodes nodes _        = Ctx (head nodes) nodes 1 (length nodes)
```

(P)rocessContext performs a transformation for all nodes of a context. The transformation is a function which accepts a context and creates a result tree fragment. The result tree fragments of all transformation steps are concatenated.

```
processContext :: Context -> (Context->[XmlTree]) -> [XmlTree]
processContext CtxEmpty _ = []
processContext ctx@(Ctx node nodeList pos len) f
     | pos > len = []
     | otherwise = f ctx ++ processContext (Ctx (nodeList!!pos) nodeList (pos+1) len ) f
```

(P)rocessContext is implemented by recursion over the position argument of the context. The incoherent use of *pos* and *pos+1* is a result of different list indexing in XSLT and Haskell. In Haskell indexing starts with zero. In XSLT it starts with one. In the last recursion step processContext is called with *Ctx ((nodelist !! len) nodelist (len+1) len)* where *nodelist !! len* evaluates to $\perp$. In other words the value of the current context node is undefined. This might sound disconcerting, but it has no consequences since the current node is never evaluated. In a language without lazy evaluation we would need to find an alternative algorithm.

A quick look back

This is a good point to hold on for a second and take a look back at what we have at this moment. We have probably implemented just a quarter of the XSLT specification yet and the hardest parts are still ahead. At this stage, the XSLT processor is still below 500 lines of Haskell code[1], but it is already able to process many useful stylesheets.

Do you remember the example from the introduction?

```
<html xsl:version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <body>
    <table>
       <xsl:for-each select="*/item">
         <tr>
           <td><xsl:value-of select="@name"/></td>
           <td><xsl:value-of select="@value"/></td>
         </tr>
       </xsl:for-each>
    </table>
  </body>
</html>
```

1 Not counting the XPath implementation.

Our current processor can actually process it. And not just that. The three *data example(s)* in the D.2 appendix of the XSLT specification can be processed after a few tiny modifications:

- In the HTML example, *xsl:sort* is not yet understood.
- In the SVG example, the *xsl:output*-tag has to be removed and the stylesheet has to be transformed to the simplified syntax by treating the *svg*-tag as a literal result stylesheet-element.
- In the VRML example, the *xsl:output*-tag has to be removed and we can directly instantiate the content of the *xsl:template*-tag, even though this is normally forbidden by the XSLT-specification.

This encouraging intermediate result justifies our bottom-up approach. From here on we can gradually increase the power of the XSLT processor. After implementing a new feature we retain a working processor, which just understands a few more meaningful stylesheets.

Copying

XSLT provides two distinct facilities to copy nodes from the input document to the output document.

```
<xsl:copy use-attribute-sets = qnames>
  <!-- content: template -->
</xsl:copy>

<xsl:copy-of select = expression />
```

The first one is used to make a shallow copy of the current node. The second one makes a deep copy of the result of an arbitrary XPath expression.

Ignoring the use-attribute-sets feature of *xsl:copy* for them moment the abstract syntax is defined as follows:

```
data Template = ...
  | TemplCopy Template
  | TemplCopyOf Expr
```

The copy-of expression is not required to return a nodeset. It can equally well return a computed number- or string-value. In these cases copy-of acts exactly like value-of.

The compilation doesn't deserve a comment:

```
compileCopy, compileCopyOf :: XmlTree -> Template
compileCopy   = TemplCopy . compileTemplate . getChildren
compileCopyOf = TemplCopyOf . parseExpr . flip fetchAttribute xsltSelect
```

The application of a copy-template is defined as follows

```
applyCopy :: Template -> Context -> [XmlTree]
applyCopy (TemplCopy template) ctx =

    -- Case 1: Root node => just use the content template
    if isRoot currNode
    then content
    -- Case 2: Any other element-node
    else if isElem currNode
    then return $ createElement name content

    -- otherwise: Just return the current node as result
    else return currNode

  where
    currNode = subtreeNT $ ctxGetNode ctx
    name     = fromJust $ getElemName currNode
    content  = applyTemplate template ctx
```

The first special case is the result of a general XPath rule: On the level of the tree-transformation no root node is ever created. However, the result of a transformation will always be a valid fragment of an XML tree (, or an error).

The second special case implements the shallow-copying of element nodes. An element node is copied without its attributes. We simply create a new element node with the same qualified name. The content template is instantiated in the same way in which it would be instantiated for the *xsl:element* template. The result of the instantiation is a sequence of attribute nodes followed by a sequence of non-attribute nodes. The former sequence is added as an attribute list to the element. Attributes appearing later in that list will override previously added attributes with the same name. The later sequence forms the child nodes of the element.

All non-element children are directly copied to the result tree.

(S)ubtreeNT converts a node of the input tree to a result tree fragment. More technically it converts from navigatable to non-navigatable trees.

The XSLT copy template is normally used within a recursive named rule. It will become more useful once we've implemented these.

The implementation of the deep-copy application is defined as follows:

```
applyCopyOf :: Template -> Context -> [XmlTree]
applyCopyOf (TemplCopyOf expr) =
    concatMap (expandRoot) . xPValue2XmlTrees . evalXPathExpr expr
  where
    expandRoot node = if isRoot node then getChildren node else return node
```

Conveniently, the XPath library already offered the *xPathValue2XmlTrees* function which transforms non-nodeset values to text-nodes. Again, special casing for the root node is required.

Creating comments and processing instructions

For the sake of completeness we will close the *gentle start* chapter with a description of the *xsl:comment* and *xsl:processing-instruction* templates.

An XML comment is an (almost) arbitrary string, contained within "<--" and "-->". In a comment "<" and ">" characters are not quoted. However, the string must not contain "--" or end in "-".

An XML processing instruction is an (almost) arbitrary string contained within *"<?Name "* and *"?>"* where *name* is a simple XML name without a colon character. However the string must not contain *"?>"*. Despite common misconceptions a processing instruction does not have an attribute list. These *attribute-list-like* strings are just a common convention, not a requirement. Therefore on the level of the XML tree a processing instruction consists of a single text node.

XSLT provides *the xsl:comment* and *xsl:processing-instruction* templates for the creation of comments and processing instruction. In both cases the content can be computed and in the case of processing instructions the name is an attribute value template and can be computed as well. Unlike elements which can be constructed from a literal syntax by literal result elements there is such syntax for processing instruction and comments.

```
<xsl:comment>
  <!-- content: template -->
</xsl:comment>

<xsl:processing-instruction name = { ncname } >
  <!-- content: template -->
</xsl:processing-instruction>
```

In the abstract syntax *xsl:comment* and *xsl:processing-instruction* are defined by:

```
data Template = ...
  | TemplComment Template
  | TemplProcInstr StringExpr Template
```

The compilation is trivial and therefore omitted. The application of both elements is defined below.

```
applyComment :: Template -> Context -> [XmlTree]
applyComment (TemplComment content) ctx =
    return $ mkCmt $ format $ collectTextnodes $ applyTemplate content ctx
  where
    format ""           = ""
    format "-"          = "- "
    format ('-':'-':xs) = '-':' ':format ('-':xs)
    format (x:xs)       = x:format xs
```

The XSLT implementation may either signal an error or apply a fix whenever the textual content of a comment doesn't satisfy the requirements stated above. Fixing of the comment is done by inserting a space between each pair of consecutive *"-"*-characters and after a terminating *"-"*. There is no strong incentive for either fixing or issuing an error. However, there are many situation in which the XSLT specification allows to either signal an error or apply a fix, ignore the offending node, etc... We usually choose to be lenient. XALAN does the same which makes it easier to test our processor by simply comparing its output to XALAN's output for the same stylesheet. If this comment-fixup is useful for other XML-applications it might be desirable to move it to the HXT core.

```
applyProcInstr :: Template -> Context -> [XmlTree]
applyProcInstr (TemplProcInstr nameExpr template) ctx =
    return $ mkXPiTree name $ format $ collectTextnodes $ applyTemplate template ctx
  where
    name = applyStringExpr nameExpr ctx
    format ""           = ""                     -- normally: format = replaceAll "?>" "? >"
    format ('?':'>':xs) = '?':' ':'>':format xs
    format (x:xs)       = x:format xs
```

For processing instruction fixup is simply done by replacing all occurrences of "?>" with "?

>". The hand-rolled implementation is tedious and should be replaced by a call to a standard Haskell function in the future. However, as of writing this, the Haskell standard libraries are lacking some very basic string manipulation facilities[2]. This will hopefully change in the future. Again, if this fixup turns out to be useful for other applications it should probably become a part of the HXT core.

(X)ml declarations cannot be created on the level of tree transformations by an XSLT-processor.

2 John Goerzen is aware of this problem and supplies a replace function and other functions missing in the standard libraries with his MissingH libraries (http://gopherproject.org/devel/missingh/html/index.html). However, these libraries are quite extensive and adding them to the HXT distribution would probably not be a good idea

Entire stylesheets

At this point the processor is able to compile and apply a single template. While this is already useful and allows many meaningful transformations, the advanced features are still missing. Within this chapter, our XSLT subset will become powerful enough to express any computable transformation on an XML document.

We show

- , how the compilation of an entire stylesheet with includes and imports can be done.
- , how named- and match- rules can be selected and applied.
- , how variables and parameters can be treated.

These are the three main topics. Other, less important features are implemented as well, but shouldn't concern us for the moment.

The compilation model

The compilation of an entire stylesheet can be decomposed into four major steps:

- Document level preprocessing. Gathers information which is local to a single stylesheet document. The result is an expanded XML tree.
- Expanding the XML tree of the current stylesheet document with XML trees of all included documents.
- Compilation of one stylesheet and its **includes**. The results are compiled stylesheet fragments like compiled rules or procedures.
- Joining of the precompiled stylesheet with its **imports**. On this level the compiled fragments of the current stylesheet are merged with the compiled imported stylesheets. The result is a compiled stylesheet.

We will discuss the first step before we expose the overall structure of the stylesheet compilation.

Document level preprocessing

Single XSLT documents have certain properties or attributes which are effective transitively for an element of a document and all its children. These attributes must be treated before the inclusion of other stylesheet documents can be performed. We can consider the following artificially condensed example:

```
<xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:extension="extension.org"
  xmlns:unused="unused.org"
  xmlns:ns1="ns1.org"
  xml:space="preserve"
  extension-element-prefixes="extension">

  <xsl:include href="inc.xsl" />
  <xsl:template match="*">
    <xsl:element name="result">
      <ns1:result/>
      <xsl:call-template name="procedure" />
    </xsl:element>
  </xsl:template>

  <xsl:template name="procedure" xmlns:ns2="ns2.org">
    <ns2:result xsl:exclude-result-prefixes="ns1">
      <inner/>
    </ns2:result>
  </xsl:template>

</xsl:transform>
```

The namespace declarations for *extension*, *xsl, unused* and *ns1* are effective for all descendants[3] of the document node. The namespace declaration for *ns2* is only effective for all descendants of the named template. The *xml:space* and *extension-element-prefixes* attributes are effective for the entire document. The *exclude-result-prefixes* attribute is only valid for all descendants of *ns2:result*. None of these attributes have any effect on the included stylesheet *inc.xsl*. Here is the output of that stylesheet with the original indentation:

```
<result>
  <ns1:result xmlns:ns1="ns1.org" xmlns:unused="unused.org"/>

<ns2:result xmlns:ns2="ns2.org" xmlns:unused="unused.org">
  <inner xmlns:ns2="ns2.org" xmlns:unused="unused.org"/>
</ns2:result>

</result>
```

The italic namespace attributes are redundant but legal. The result element must not have any namespace attributes. Both *ns1:result* and *ns2:result* must have the *xmlns:unused* attibute, even though none of the created elements refers to it. *(N)s2:result* and *inner* must not declare an *xmlns:ns2* attribute, as it is marked as an excluded result namespace in their context. These rules might seem arbitrary, but they are justified as it is not always obvious which namespaces are actually used within an XML document. For example if we use XSLT to transform XSLT documents, there are many possible situations in which namespace bindings are required, but not used in element- and attribute names.

A simple and effective solution to deal with these transitive features is to expand them entirely, so that any element in the stylesheet XML-tree is provided with all effective attributes. This approach might appear to be prohibitive, but Haskell's lazyness prevents us from evaluating most of these redundant attributes. The general procedure for document level preprocessing is given below.

```
prepareXSLTDocument :: XmlTree -> XmlTree
prepareXSLTDocument  = expandExEx . expandNSDecls . stripStylesheet . removePiCmt
```

We will defer the discussion of whitespace stripping. It is addressed in detail on page 58 ff., where we have more of the machinery at hands to implement it.

Removal of precessing instructions and comments

Comments and processing instructions within stylesheet documents are not interpreted by XSLT and can therefore be removed. This can be done by a simple filter operation:

3 We mean the reflexive and transitive closure of a node and its children by *descendants*.

```
removePiCmt :: XmlTree -> XmlTree
removePiCmt = fromJustErr "XSLT: No root element" . filterTree (\n -> not (isPi n) && not (isCmt n))
```

Namespace expansion

The *expandNSDecls* procedure expands a document of the form

```
<result a="1" xsl:version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <inner b="2" xmlns:new="new.ns" />
</result>
```

to a document of the following form (at least conceptually, the namespace bindings for *xml* and *xmlns* can be omitted):

```
<result a="1" xsl:version="1.0" xmlns:xsl  ="http://www.w3.org/1999/XSL/Transform"
                                xmlns:xml  ="http://www.w3.org/XML/1998/namespaces"
                                xmlns:xmlns="http://www.w3.org/2000/xmlns/" >
  <inner b="2" xmlns:new="new.ns"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
               xmlns:xml  ="http://www.w3.org/XML/1998/namespaces"
               xmlns:xmlns="http://www.w3.org/2000/xmlns/" />
</result>
```

The careful reader might have noticed the inconsistent conventions for the *xml* and *xmlns* namespace URIs. One has a trailing slash, the other doesn't have it. However, the particular spelling is required in both cases. To extract the namespace prefix to namespace URI mappings of an XML node we have a function:

`getUriMap :: XmlNode n => n -> UriMapping` where: `type UriMapping = Map String String`

The usage of the type class *XmlNode* allows us to use *getUriMap* on both *XmlTree* and *XNode.*

Expansion of exclusion and extension markers

The *xsl:exclude-result-prefixes* or *xsl:extension-element-prefixes* on literal result elements and the unprefixed equivalents on the stylesheet element are expanded in this compilation step. The following procedure iterates over the entire stylesheet document tree:

```
expandExEx :: XmlTree -> XmlTree
expandExEx = mapTreeCtx expandExExElem ([xsltUri,xmlNamespace,xmlnsNamespace],[])
```

Initially the *xml*, *xmlns* and *xsl* namespaces are marked as excluded namespaces. The processing of an entire tree can be implemented in terms of higher order functions.

```
mapTreeCtx :: Tree t => (c -> a -> (c, b)) -> c -> t a -> t b
mapTreeCtx f c tree =
    mkTree b $ map (mapTreeCtx f cN) $ getChildren tree
  where
    (cN, b) = f c $ getNode tree
```

The mapping operation works like a *normal* map. It creates a tree with an identical structure but transformed node values. Additionally a context is passed which can be used to accumulate contextual information from the higher levels of the tree hierarchy. The example below demonstrates how *mapTreeCtx* can be used to transform a tree with relative directory paths to a tree with absolute directory paths.

```
> mapTreeCtx (\c n -> let p = c ++ "/" ++ n in (p,p)) ""
     $ mkTree "a" [mkTree "b" [mkLeaf "c"], mkLeaf "d"]
---"/a"
   |
   +---"/a/b"
   |   |
   |   +---"/a/b/c"
   |
   +---"/a/d"
```

The paper *Origami programming* by Jeremy Gibbons which appears in *the fun of programming* gives many beautiful examples on how tree operations can be expressed in terms of higher order functions.

The effective exclusion and extension markers for a single node are computed as follows:

```
expandExExElem :: ([String], [String]) -> XNode -> (([String], [String]), XNode)
expandExExElem c@(excl, ext) node
  | isElem node = ((exclAcc, extAcc), nodeNew)
  | otherwise   = (c, node)
  where
    nodeNew    = setAttribute nameExcl (unwords exclAcc)
                 $ setAttribute nameExt (unwords extAcc) node
    exclAcc    = exclNew ++ excl
    extAcc     = extNew  ++ ext
    exclNew    = extNew  ++ (parsePreList><node $ fetchAttributeWDefault node nameExcl "")
    extNew     =             parsePreList><node $ fetchAttributeWDefault node nameExt  ""
    (nameExcl,
     nameExt)  = if (namespaceUri $ fromJust $ getElemName node) == xsltUri
                   then (xsltExlcudeResultPrefixes   , xsltExtensionElementPrefixes   )
                   else (xsltExlcudeResultPrefixesLRE, xsltExtensionElementPrefixesLRE)
```

XSLT instructions use the unprefixed attribute names while literal result elements use the prefixed forms. We are a bit more lenient here than the specification requires. We do accept these attributes on any XSLT element and not just on the stylesheet document; however, the behavior we implemented is the behavior required by the XSLT 2.0 specification. The exclusion namespace URIs automatically contain the extension namespace URIs. The lists of whitespace separated prefixes are transformed to lists of whitespace separated namespace URIs according to the namespace bindings of the current node. The notation *parsePreList><node* is just a shortcut for:

```
parsePreList (getUriMap node)
```

more general:

```
infixl 9 ><
(><) :: XmlNode n => (UriMapping -> a ) -> n -> a
f><node = f $ getUriMap node
```

f><node should be read as "*f" is evaluated in the context of "node"*. *ParsePreList* is:

```
parsePreList :: UriMapping -> String -> [String]
parsePreList uris = map (lookupPrefix uris) . words
```

We can use the expanded exclusion marker to evaluate all namespace mappings which must be added to a literal result elements:

```
extractAddUris :: XmlTree -> UriMapping
extractAddUris node =
    (Map.filter (`notElem` exclUris))><node
  where
```

```
    exclUris = words $ fetchAttributeWDefault node xsltExlcudeResultPrefixesLRE ""
```

Alternatively we could have implemented document level preprocessing with attributed XML trees. It is possible to define a type class for attributed XML nodes, so that instances of this class can be used whenever an instance of the *XmlNode* class is required. The idea seems charming as it would allow problem specific data structures and thereby stricter typing and probably more efficient lookup than the generic XML tree structure. Actually there is an implementation of the XSLT preprocessor with attributed XML nodes; however, the benefits of this approach seem to be outweighed by the complexity which is added by the different types of XML trees.

Includes and imports

Let us consider a full blown XSLT stylesheet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:import href = "imp_1.xsl" />
  <!-- other imports -->
  <xsl:import href = "imp_k.xsl" />

  <!-- some nifty rules and stuff. -->

  <xsl:include href = "inc_1.xsl" />

  <!-- some more clever rules. -->

  <xsl:include href = "inc_2.xsl" />

  <!-- yet more ingenious stuff -->

</xsl:stylesheet>
```

The full blown stylesheet starts with a list of *xsl:import* elements. These are followed by a list of other top level elements and *xsl:include* elements in an arbitrary but semantically meaningful order. All elements which are allowed as children of an *xsl:stylesheet* or synonymously *xsl:transform* element are called top level elements. The most prominent are named and match rules, global variables and stylesheet parameters.

Includes are inserted into the stylesheet on the level of XML trees. The XSLT include element is therefore comparable to the include facility of the C preprocessor. On the other hand **imports** can be regarded as separate stylesheets. These are comparable to modules in Haskell. In particular, imports can be used to control the priority of rules.

Any XSLT processor must (at least conceptually) ensure that
- , an XML tree is properly identified as a stylesheet.
- , an XML tree using the simplified syntax is identified as a stylesheet.
- , includes are properly inserted into the stylesheet XML-tree.
- , includes using the simplified syntax are properly expanded.
- , imports are compiled separately.
- , recursive imports and includes are detected.
- , the previously discussed document level preprocessing is properly performed.

For the outside world the stylesheet compilation is triggered by the following function:

```
compileStylesheet :: XmlTree -> IO CompiledStylesheet
compileStylesheet = compileStylesheetWIncStk [] . prepareXSLTDocument
```

After the document level preprocessing an empty stack with the URIs of all previously included or imported stylesheets is created. This stack (or list) is necessary to detect recursive includes or imports.

Interestingly, *compileStylesheet*'s return type is not a *CompiledStylesheet*, but an IO-command of the type *CompiledStylesheet*. This is a result of Haskell's pureness. The result of the *compileStylesheet* function is dependent on the outside world, as include- and import files must be fetched from the file system or even external servers. There is no way to guarantee that *compileStylesheet* will always return the same compiled stylesheet for the same input. However, *compileStylesheet* will always return the same IO command for the same input. The compiled stylesheet can only be manipulated from within this IO command. This *monadic* style of programming usually causes severe headaches in the beginning. We will try to reduce the need for this programming style to the bare minimum and delegate (almost) all the real work to pure functions.

The central compilation routine is defined as follows:

```
compileStylesheetWIncStk :: [String] -> XmlTree -> IO CompiledStylesheet
compileStylesheetWIncStk incstack node =

  -- ======= 1: simplified syntax
  if isLREstylesheet xslNode then
    return $ assembleStylesheet (lre2stylesheet xslNode) []

  -- ======= 2: regular syntax
  else if isStylesheetElem xslNode then
    do
      -- ======= 2.1: gather included stylesheets
      expandedContent <- expandIncludes incstack content

      -- ======= 2.2: compile imported stylesheets
      (imps, rest) <- return $ partition (isElemType xsltImport) expandedContent
      imports      <- mapM (compileStylesheetFromUriWIncStk incstack . getHRef) $ imps

      -- ======= 2.3: compile content
      expandedStylesheet <- return $ setChildren rest xslNode
      return $ assembleStylesheet expandedStylesheet imports

  -- ======= 3: unknown document type
  else error "Expected: Either xsl:stylesheet/xsl:transform or simplified syntax"

  where
    content     = getChildren xslNode
    (xslNode:_) = filter isElem $ getChildren $ node
    getHRef     = flip fetchAttribute xsltHRef
```

1 - Simplified syntax:

While we have previously been able to process stylesheets written in the simplified syntax, we must now properly integrate them in the general compilation process.

A literal result element must have an *xsl:version* attribute to identify itself as a simplified syntax stylesheet:

```
isLREstylesheet :: XmlTree -> Bool
isLREstylesheet = flip hasAttribute xsltVersionLRE
```

The stylesheet:

```
<lre xsl:version="1.0" xmlns:xsl="..."> ... </lre>
```

must be treated like the equivalent stylesheet:

```
<xsl:stylesheet version="1.0" xmlns:xsl="...">
  <xsl:template match="/">
     <lre> ... </lre>
  </xsl:template match>
</xsl:stylesheet>
```

This can be implemented by transformations on the level of XML trees:

```
lre2stylesheet :: XmlTree -> XmlTree
lre2stylesheet = mkElement xsltTransform [] . return . lre2template
```

where:

```
lre2template :: XmlTree -> XmlTree
lre2template = mkElement xsltTemplate [mkAttr xsltMatch [mkText "/"]] . return
```

2 - Regular syntax:

The document node of a stylesheet in regular syntax must be either *xsl:stylesheet* or *xsl:transform* and it must have a *version* attribute:

```
isStylesheetElem :: XmlTree -> Bool
isStylesheetElem node =
  (isElemType xsltTransform node || isElemType xsltStylesheet node) && hasAttribute node xsltVersion
```

2.1 - Gathering includes:

Includes must be collected before the *"real"* compilation starts. In particular, *includes* are collected before the *imports* are compiled, since they (*the includes*) might contain further *imports*.

```
expandIncludes :: [String] -> [XmlTree] -> IO [XmlTree]
expandIncludes incstack = liftM concat . mapM (expandInclude incstack) . filter isElem
```

(E)xpandIncludes tries to expand any element node within a stylesheet XML tree. Non-element nodes below the stylesheet element must be ignored. The function could be expressed a bit more concisely if we had previously defined *concatMapM f = liftM concat . mapM f*. However, we use this expression only once and don't want to produce too many utility functions.

```
expandInclude :: [String] -> XmlTree -> IO [XmlTree]
expandInclude incstack node =
  if isElemType xsltInclude node
    then
    do
       -- ======= read include-stylesheet and extract stylesheet node
       href        <- return $ fetchAttribute node xsltHRef
       docNode     <- readStylesheetWIncStk incstack href
       (xslNode:_) <- return $ filter isElem $ getChildren docNode

       -- ======= check for simplified syntax
       if isLREstylesheet xslNode
         then return [lre2template xslNode]

       -- ======= check for xsl:stylesheet or xsl:transform
         else if isStylesheetElem xslNode
                then expandIncludes (href:incstack) $ getChildren xslNode

       -- ======= include file has an unknown type
```

```
              else error $ "Error: Included file " ++ href ++ " is not a stylesheet"
    else return [node]
```

Side note: The somewhat special formatting of the if-expression is needed. In the stylesheet function we had an if-expression in which a do-block was nested. Now we have a do-block in which an if-expression is nested. Within a do-block the compiler (conceptually) tries to find monadic commands which are then transformed to an appropriate sequence of a >>=, >> and lambda expressions. Without sufficiently deep indentation then or else parts are interpreted as broken commands and will lead to incomprehensible error messages.

(E)xpandInclude tests if the current node is an *xsl:include* element. If it is not, a list containing just the node is returned. If it is, the *href* attribute is read. Fortunately, this attribute is a literal and not an expression.

The included stylesheet is read as follows:

```
readStylesheetWIncStk :: [String] -> String -> IO XmlTree
readStylesheetWIncStk incstack uri =
  if uri `elem` incstack
  then error $ "Error: " ++ uri ++ " is recursively imported/included."
            ++ concatMap ("\n  imported/included from: " ++) incstack
  else readDocumentIO [(a_preserve_comment, "0")] uri >>= return . prepareXSLTDocument
```

If a recursive include or import is detected this function will return a nicely formatted error, like:

```
inc2.xsl is recursively imported/included.
  imported/included from: inc3.xsl
  imported/included from: inc2.xsl
  imported/included from: inc1.xsl
  imported/included from: global.xsl
```

A stylesheet document can always be read without its comments since comments within a stylesheet (, as opposed to comments within an input document) have no meaning in XSLT. Whitespace-stripping is performed before any further processing.

The result tree is checked for the simplified syntax at first. If it is detected, the literal result element is expanded to an xsl:template which matches the root node.

If the result is a conventional stylesheet *expandInclude* pushes the URI of the included stylesheet on the include-stack and (indirectly recursively) expand the child-nodes of the xsl:element.

<u>2.2 - compile imported stylesheets</u>

After the stylesheet has been expanded on the tree level. Its children are partioned into xsl:import elements and the rest. The imported stylesheets are compiled completely separate:

```
compileStylesheetFromUriWIncStk :: [String] -> String -> IO CompiledStylesheet
compileStylesheetFromUriWIncStk incstack uri =
  readStylesheetWIncStk incstack uri >>= compileStylesheetWIncStk (uri:incstack)
```

For convenience the XPath processor exports the following function which isn't used internally:

```
compileStylesheetFromUri :: String -> IO CompiledStylesheet
compileStylesheetFromUri = compileStylesheetFromUriWIncStk []
```

2.3 - Triggering the compilation of the stylesheet content:

The expanded content is appended to the stylesheet node. *(A)ssembleStylesheet* is provided with the expanded stylesheet (an XML tree) and the imorted stylesheets (an in-order list of compiled stylesheets). At the moment there is nothing we can do here. We return to that function when are able to compile a top level element.

Matching

Before we can discuss rules, we have to examine how a node can be matched against a pattern. So far we have used XPath expressions to retrieve and compute values of different types. Basically, these expressions selected nodes and combined the results, possibly converting them to strings, numbers or booleans. However, XSLT needs to be able to check whether a given node (the context node) matches a pattern. Syntactically, patterns are a subset of expressions. The matching of patterns can be understood in terms of selecting nodes, as we will show below. Patterns have a priority. In general patterns that match fewer nodes have a higher priority. However, there are exceptions.

There is some tension if

- , patterns should be implemented as a part of the XPath library.
- , patterns should be implemented in terms of the XPath library.
- , patterns should be implemented independently.

The first and second option create some tangling between the XPath and XSLT modules, whereas the third option leads to code duplication. We will implement patterns mostly in terms of the XPath library. However, we need to make some additions to it to be able to do so.

Patterns as a subset of expressions:

The XSLT specification gives a grammar for patterns, which we do not repeat here. Likewise the XPath specification gives a grammar for XPath expressions which we do not repeat here.

Torben Kuseler translates the XPath grammar to an abstract syntax for XPath expressions in Haskell. Based on Torben's abstract syntax, we can check whether an XPath expression is a pattern (, or a *match*-expression):

```
isMatchExpr :: Expr -> Bool
isMatchExpr (GenExpr Union exprs)                          = all isMatchExpr exprs
isMatchExpr (PathExpr _ _)                                 = True
isMatchExpr (FctExpr "id" [LiteralExpr _])                 = True
isMatchExpr (FctExpr "key" [LiteralExpr _, LiteralExpr _]) = True
isMatchExpr _                                              = False
```

Unions (e.g. : "a|b") are patterns, iff all of their terms are patterns. Path-expressions (eg. *a*, *a/b* ...) are generally considered to be patterns. This is probably a bit too lenient, but good enough for now. Besides unions and paths only the two function expressions *key* and *id* are allowed. Both are allowed only if all of their arguments are literal strings. Other functions, literal strings and literal numbers are not allowed as patterns.

Similar to the the previously discussed types of XPath expressions, we introduce a new type for patterns:

```
newtype MatchExpr = MatchExpr Expr
```

The parsing of a match expression is defined as follows:

```
parseMatch :: UriMapping -> String -> MatchExpr
parseMatch uris str =
    if isMatchExpr expr
    then MatchExpr expr
    else error $ str ++ " is not a legal match-expression"
  where
    expr = parseExpr uris str
```

It's helpful to perform the checking within this function, as we have access to the literal expression here. Therefore, we can produce better error messages. The namespace prefix to namespace URI mappings are passed to the XPath parser, so that it can properly expand qualified names within expressions. We had omitted this property in the introductory chapter, since it requires document level preprocessing as described on page 24 ff.

Priorities of patterns:

A union is treated as several patterns of which each can have a distinct priority. Therefore we implement:

```
splitExpr :: Expr -> [Expr]
splitExpr (GenExpr Union expr) = expr
splitExpr rest                 = [rest]
```

It's useful to define the inverse functions as well:

```
unionExpr :: [Expr] -> Expr
unionExpr [e] = e
unionExpr es  = GenExpr Union es
```

The priority of a single expression can be computed as follows:

```
computePriority :: Expr -> Float
computePriority (PathExpr Nothing (Just (LocPath Rel [Step _ ntest []])))
                = computeNTestPriority ntest
computePriority _ = 0.5
```

Basically a path consists of a set of restrictions (like in: *document[self::*]*) and a relative or absolute location path with one (*document*) or several location-steps (*document/chapter*). Each step has an axis-specifier, like child (*child::chapter* usually written as *chapter*) or attribute (*attribute::nr* usually written as *@nr*). The axis-specifier does not affect the priority and can therefore be ignored (_).

The priority 0.5 is assigned to any path which is not a relative location path, has more than one step or any restrictions. Examples are: */*, */a*, *a/b* or *a[self::*]*. The *key('lit','lit')* and *id('lit')* patterns have priority 0.5 as well. Interestingly, XSLT 2.0 makes a special case for the root node */*. It has the priority -0.5. From the XSLT 1.0 rules the priority 0.5 can be derived. We can check this experimentally with Xalan as well.

All that's left is the calculation of the priority of a single node test:

```
computeNTestPriority :: NodeTest -> Float
computeNTestPriority (PI _)        =  0.0
computeNTestPriority (TypeTest _)  = -0.5
computeNTestPriority (NameTest nt)
  | namePrefix nt /= ""
    && localPart nt == "*"        = -0.25
  | localPart nt == "*"           = -0.5
  | otherwise                     =  0.0
```

Processing instructions matched with *processing-instruction('name')* receive the priority *0.0*. The name must be a literal. There are no wildcards allowed. Type tests like *processing-instruction()*, *comment()* or *text()* receive the priority *-0.5*. Likewise the patterns *** or *@** receive the priority *-0.5*. *namespace:** patterns have the priority *-0.25*. The most common type of patterns: *name*, *ns:name*, *@attr*, *@ns:attr* all receive the priority *0.0*. XPath 1.0 does not allow patterns of the form **:name*. In XPath 2.0 these patterns would have the same priority as the *ns:** patterns which is *-0.25*.

While usually a higher priority indicates that a pattern matches fewer nodes, this is not necessarily the case. As an example **[self::*]* has the priority *0.5* whereas the more specialized pattern *ns:document* has the priority 0.0.

Intelligent splitting:

As already mentioned a pattern can have more than one priority, if it consists of a union of alternative branches separated by "|". We want to give an algorithm to split such a pattern into as few as possible sub patterns with different priorities.

For example:

```
"*"                 is splitted to      "*"            with priority -0.5
"a"                 is splitted to      "a"            with priority  0.0
"@a"                is splitted to      "@a"           with priority  0.0
"a/b"               is splitted to      "a/b"          with priority  0.5
"a|b|c"             is splitted to      "a|b|c"        with priority  0.0
"a|b/c|d|e/f"       is splitted to      "a|d"          with priority  0.0
                               and      "b/c|e/f"      with priority  0.5
"@*|ns:*|*|@ns:*"   is splitted to      "@*|*"         with priority -0.5
                               and      "ns:*|@ns:*"   with priority -0.25
"*|ns:*|a|a/b"      is splitted to      "*"            with priority -0.5
                               and      "ns:*"         with priority -0.25
                               and      "a"            with priority  0.0
                               and      "a/b"          with priority  0.5
```

This is done by the following Haskell function:

```
splitMatchByPrio :: Expr -> [(Float, Expr)]
splitMatchByPrio =
    map compress . groupBy eq . sortBy cmp . map (computePriority &&& id) . splitExpr
  where
    eq  x y  = fst x == fst y
    cmp x y  = compare (fst x) (fst y)
    compress = (head *** unionExpr) . unzip
```

At first the expression is split into all its subexpressions (for the sake of readability we will treat expressions as strings here):

```
splitExpr "a|b/c|d|e/f" -> ["a","b/c","d","e/f"]
```

Next, priority/expression tuples are build:

```
map (computePriority &&& id) ["a","b/c","d","e/f"] -> [(0.0,"a"),(0.5,"b/c"),(0.0,"d"),(0.5,"e/f")]
```

Then these tuples are sorted by priority:

```
sortBy cmp [...] -> [(0.0,"a"),(0.0,"d"),(0.5,"b/c"),(0.5,"e/f")]
```

After that the sorted tuples are grouped by equal priority:

```
groupBy eq [...] -> [ [(0.0,"a"),(0.0,"d")], [(0.5,"b/c"),(0.5,"e/f")] ]
```

And finally the groups are merged again:

```
map compress [[...]] -> [(0.0,"a|d"), (0.5,"b/c|e/f")]
```

The use of the &&& and *** operators might seem a bit strange here. These function are provided by the Arrow class, which generalizes functions[4] and provides (among others) some operators to compose functions on tuples in a point-free style. *Normal*-functions are just a special instance of the Arrow class. For them the two operations can be defined as follows:

```
instance Arrow (->) where
  ...
  f &&& g = (\x      -> (f x, g x))
  f *** g = (\(x, y) -> (f x, g y))
```

In his book *Introduction to functional programming using Haskell* Richard Bird defines a function pair which is equivalent to *uncurry (&&&)* and a function cross which is equivalent to *uncurry (***)* and uses them to define functions like unzip in a very concise manner. We would have preferred to write pair and cross instead of &&& and ***. However, &&& and *** are what the standard libraries offer.

<u>*Application of patterns:*</u>

The application of match-expression can be understood in terms of the application of select-expressions. The XSLT specification states:

A node matches a pattern if the node is a member of the result of evaluating the pattern as an expression with respect to some possible context; the possible contexts are those whose context node is the node being matched or one of its ancestors.

This can be implemented almost literally:

4 Actually *arrow* is a term from category theory. We could say that arrows are to category theory what functions are to set theory. However, it shouldn't really bother us here. Just think of functions and composition functions like (.) or ($). Within this framework we just use a few new composition functions like (***) and (&&&). The Haskell notations are often far more esoteric than the equivalent notations in category theory. For example f *** g would be written as $f \times g$ and f &&& g as $\langle f, g \rangle$ in category theory. Notations like f `cross` g and f `pair` g might have been a better choice.

```
applyMatch :: MatchExpr -> Context -> Bool
applyMatch (MatchExpr expr) ctx =
    matchBySelect (SelectExpr expr) (ctxGetNode ctx) ctx
  where
    matchBySelect :: SelectExpr -> NavXmlTree -> Context -> Bool
    matchBySelect _ _ CtxEmpty = False
    matchBySelect expr matchNode ctx =
       if matchNode `isNotInNodeList` applySelect expr ctx
       then matchBySelect expr matchNode $ ctxSetNodes (maybeToList $ upNT $ ctxGetNode ctx) ctx
       else True
```

We start to match by using the match node as the context node of the selection. If we are successful the match node will be within the selected nodes and we have found a match. If not, the current node moves one up on the ancestor axis and we repeat this procedure. Since the root node has no ancestors the context will be set to *CtxEmpty* before the last recursion step. This allows the recursion to terminate and return *False*.

Possible optimizations:

Unfortunately this implementation is not particularly efficient. In case of a non-match, many nodes are potentially selected and tested. Some optimizations are possible, even if we don't implement matching fully separately from the selection. One simple and yet powerful optimization would be to provide special casing for the most common cases. In particular *name*, *@name*, * and *@** could be treated by a specialized function. In the general case we would still rely on *matchBySelect*. However, this implementation allowed us to have something working and meaningful to quickly enhance the expressive power of our XSLT processor and to be able to verify test cases. We might consider it as an executable specification. It has priority to optimize the matching algorithm, to make our implementation more useful for actual applications which have strict performance requirements.

Related to the tension patterns create between the XPath and XSLT specifications, let's consider the following approach. Imagine XPath defined a *match* function, which would be true whenever the current context node could be matched against the argument-expression and false otherwise. This would make it easier to define XSLT matching in terms of XPath expressions and therefore reduce the tangling between the specifications. And it would allow to us to express matching locally within *xsl:for-each* in a concise fashion:

```
<xsl:for-each select="*/*/*|*/*/*/@*">
  ...
  <xsl:if test="match(/a/b/c)">
    Match: /a/b/c
  </xsl:if>
  <xsl:if test="match(a/@b)">
    Match: a/@b
  </xsl:if>
  ...
</xsl:for-each>
```

Instead of the unwieldy:

```
<xsl:for-each select="*/*/*|*/*/*/@*">
  ...
  <xsl:if test="self::c[parent::b[parent::a[not(parent::*)]]]">
    Match: /a/b/c
  </xsl:if>
  <xsl:if test="not(self::*) and not(processing-instruction()) and name()='b' and parent::a">
    Match: a/@b
  </xsl:if>
  ...
</xsl:for-each>
```

Which leads us to the question: Can we always express matching in terms of node tests like in the example above? Unfortunately the answer seems to be no: It doesn't seem to be possible to transform the key and id patterns to equivalent node tests. Also, testing whether a node is an attribute can only be done by exclusion and without a normal name test as the example demonstrates. However, this transformation could be an optimization for a large set of match-patterns.

Rules

Now we have got all the machinery in place to implement the most powerful and (unfortunately) most complicated feature of the XSLT language: Rules, or *xsl:template*'s as you might call them.

The syntax for *xsl:template* is:

```
<xsl:template
  match = pattern
  name = qname
  priority = number
  mode = qname>
  <!-- content: xsl:param*, template -->
</xsl:template>
```

Any of the attributes of a rule can be considered to be optional. However, an *xsl:template* must have at least the match or the name attribute, but can have both. These template rules have two different purposes: They can be used as named procedures ***and*** they can be used as rules which are selected implicitly by a matching algorithm.

Normalization and compilation of rules

In effect one *xsl:template* can have a variety of different purposes as the example below illustrates:

```
<xsl:template match="a|b/c|d|e/f" name="proc1" mode="m">
  params, content
</xsl:template>
```

The template above can be called explicitly as a procedure with the name *proc1*. It can also be implicitly selected as a best match for a node in the mode *m* (we come to modes later). However, even if it is selected implicitly, parameters can be passed. To make the confusion perfect, the sub-patterns "a" and "d" on the one hand and "b/c" and "e/f" on the other hand match with different priorities, 0.0 and 0.5 respectively. It is hard to say how well XSLT programmers get along with this sort of overloaded meanings. However, to make rules more tractable for the XSLT processor we will apply certain normalization steps on the level of the abstract syntax. In effect the rule above will be compiled as if it was decomposed like below:

```
<!-- named rule : -->
<xsl:template name="proc1">
  params, content
</xsl:template>
```

```
<!-- match rules : -->
<xsl:template match="a|d" priority="0.0" mode="m">
  params, content
</xsl:template>

<xsl:template match="b/c|e/f" priority="0.5" mode="m">
  params, content
</xsl:template>
```

Now we can define match- and named rules separately. In the later case we have:

```
data NamedRule = NamRule ExName [Variable] Template
```

The first attribute is the procedure name which is stored as an expanded name. Unlike a qualified name an expanded name is unique representation of an name and can therefore be used as a key. The second refers to the list of parameters (we come to that later) and the third is the procedure body.

Match rules are defined below:

```
data MatchRule = MatRule MatchExpr Float (Maybe ExName) [MatchRule] [Variable] Template
```

Here the first argument is the match expression. The second is the priority. The third is the mode (Nothing for the default mode). The fourth is the list of all imported match rules. These are needed when we implement *xsl:apply-imports*. The fifth represents the parameter list and the last refers to the rule body.

The commonalities between match- and named rules can be moved to a type class:

```
class Rule a where
  getRuleContent :: a -> Template
  getRuleParams  :: a -> [Variable]
```

where:

```
instance Rule NamedRule where
  getRuleContent (NamRule _ _ c) = c
  getRuleParams  (NamRule _ p _) = p
```

and:

```
instance Rule MatchRule where
  getRuleContent (MatRule _ _ _ _ _ c) = c
  getRuleParams  (MatRule _ _ _ _ p _) = p
```

A single *xsl:template* element is compiled to zero or one named rule and zero to four match rules. However, it will always compile to at least one named- or one match rule. Therefore the compileRule function has a tuple of an optional named rule and a list of match rules as its return type. The first argument is a list of all imported match rules ordered by descending import precedence and priority. This list is needed for the *xsl:apply-imports* instruction (an equivalent to Java's *super*.foo(), we come to that later). We will show later how it is obtained.

```
compileRule :: [MatchRule] -> XmlTree -> (Maybe NamedRule, [MatchRule])
compileRule imports node =

    if isNothing match && isNothing name
    then error "Error: Bogus rule (xsl:template) with neither match nor name attribute is illegal"
    else if isJust mode && isNothing match
    then error "Error: Bogus mode attribute on non-match rule is illegal"

    else if isJust priority && isNothing match
    then error "Error: Bogus priority attribute on non-match rule is illegal"

    else
      (
        liftM (\name -> NamRule name params template) name
      , concat $ maybeToList $ liftM (assembleMatchRule priority mode imports params template) match
      )

  where
    match      = liftM (parseMatch><node)  $ tryFetchAttribute node xsltMatch
    name       = liftM (parseExName><node) $ tryFetchAttribute node xsltName
    priority   = liftM read                $ tryFetchAttribute node xsltPriority
    mode       = liftM (parseExName><node) $ tryFetchAttribute node xsltMode
    template   = compileTemplate content
    params     = map compileVariable paramsXml
    (paramsXml, content) =
                  partition (isElemType xsltParam) $ getChildren node
```

Some sanity checking has to be done. Rules without match _or_ name attributes are rejected. Likewise a mode or a priority is only meaningful for match-rules. Pure procedures with modes or priorities are rejected. Since any of the rules attributes is optional it will be conditionally transformed to the respective internal representation (a match-expression, an expanded name or a float). *LiftM* on the Maybe monad can be defined by *LiftM f = maybe Nothing (Just . f)*. It lifts a conventional function to a function which accepts and returns maybes.

(A)ssembleMatchRule creates exactly one match-rule if the priority attribute is set. Otherwise the expression is split into subexpressions with equal priority and for each of these subexpression/priority tuples one match-rule is created.

```
assembleMatchRule :: Maybe Float
                  -> Maybe ExName
                  -> [MatchRule]
                  -> [Variable]
                  -> Template
                  -> MatchExpr
                  -> [MatchRule]
assembleMatchRule pri m imp par tmpl mtch@(MatchExpr expr) =
    if isJust pri
    then return $ MatRule mtch (fromJust pri) m imp par tmpl
    else map expand $ splitMatchByPrio expr
  where
    expand (pri, mtch) = MatRule (MatchExpr mtch) pri m imp par tmpl
```

The function *splitMatchByPrio* which performs the intelligent splitting of an expression has been explained in the last chapter.

Import precedence

Imports precedence in XSLT means that:

- Elements within the expanded current stylesheet take precedence over elements within all stylesheets it imports. The expanded stylesheet consists of the current stylesheet and the transitive closure of its includes.
- An imported stylesheet takes precedence over any other imported stylesheet which has been imported previously in document order of the expanded stylesheet.
- Import precedence is stronger than any other kind of precedence.

Within our compilation model import precedence is a dynamic property. Although we compile imported stylesheets separately from each other, they can refer to variables, match-rules and procedures which are not visible in the current stylesheet.

Consider the following example:

```
A imports B
          B imports C
          B imports D
A imports E
          E imports F
```

Imagine somewhere in stylesheet C a variable reference to V_1 appears. Both C and F define a variable with that name. The reference in C will be *dynamically* bound to the definition in F because F has a higher import precedence than C. It doesn't matter that F's definition of V_1 is not visible when compiling C. Import precedence is a strict weak ordering. In our example we have: *C<D<B<F<E<A*.

You might ask why the import precedence is discussed here and not in the previous chapter on the compilation model. The answer is that this is the first time we actually need it. We will not define a data type for import precedence and there is no general mechanism which deals with import precedence. The trick is simply to construct the data structures for named rules, match rules and other elements in such a way, that the import precedence is implicitly maintained.

```
assembleRules :: [XmlTree] -> [MatchRule] -> [Map ExName NamedRule]
                  -> (Map ExName NamedRule, [MatchRule])
assembleRules nodes importedMatches importedProcs =
    (resProcs, resMatches)
  where

  -- matches:
    resMatches        = localMatches ++ importedMatches
    localMatches      = reverse $ sortBy cmp matches
    cmp rulA rulB     = compare (getRulePrio rulA) (getRulePrio rulB)

  -- procedures:
    resProcs          = Map.unions (localProcs:importedProcs)
    localProcs        = foldl ins Map.empty procs
    ins map rule      = Map.insertWith (error $ "named-rule "++ show (getRuleName rule)
                                        ++" is already defined on this level")
                                (getRuleName rule) rule map

  -- compile all xsl:template's:
    (procs, matches) = catMaybes *** concat $ unzip $ map (compileRule importedMatches) nodes
```

(A)ssembleRules receives a list of all imported match rules, ordered by descending precedence and priority. A list of mappings from procedure names to named-rules is provided as well. Each map belongs to one imported stylesheet and the list is ordered by descending precedence. All match-rules are ordered by descending priority. Match rules with the same priority are ordered in reverse document order. That is, if two match rules both match a node and have the same priority and precedence, the one appearing last in document order is selected. With this ordering the first matching rule in the result-list is always the best match. The situation is simpler with named rules. Within the map, there just one element for each key. If the same procedure is defined twice within an expanded stylesheet, an error is issued. A left-biased union is build from the maps of the current stylesheets and each of its imports. That is if *CS* is the current stylesheet and I_n to I_1 are the imports ordered by descending priority, than a procedure from I_k is only added to the union $CS \cup I_n \cup ... \cup I_1$ if there is no procedure with the same name in the union

$CS \cup I_n \cup \ldots \cup I_{k+1}$.

With the machinery we have at our hands by now, it is possible to give a preliminary but meaningful definitions for the previously omitted *CompiledStyledsheet* data type and the *assembleStylesheet* function.

For now the compiled stylesheet just consists of all named-rules and all match-rules:

```
data CompiledStylesheet = CompStylesheet [MatchRule] (Map ExName NamedRule)

getMatchRules :: CompiledStylesheet -> [MatchRule]
getNamedRules :: CompiledStylesheet -> (Map ExName NamedRule)
```

(A)ssembleStylesheet can now be defined as follows:

```
assembleStylesheet :: XmlTree -> [CompiledStylesheet] -> CompiledStylesheet
assembleStylesheet xslNode imports=
    CompStylesheet matchRules namedRules
  where
    -- compiled contents:
    (namedRules,
     matchRules)          = assembleRules ruleElems importedMatchRules importedNamedRules

    -- element content:
    (ruleElems, rest)     = partition (isElemType xsltTemplate) $ getChildren xslNode

    -- imported stuff:
    importedNamedRules    = map getNamedRules revImports
    importedMatchRules    = concatMap getMatchRules revImports
    revImports            = reverse imports
```

The functions *getNamedRules* and *getMatchRules* on compiled stylesheets can be trivially implemented. Reversing the imports ensures descending import precedence which is helpful for most cases. Later, the element children of the stylesheet element will be gradually reduced by a series of calls to partition until all understood top level elements are compiled.

Application of a match rule

Basically, match rules are implicitly applied when the stylesheet is first instantiated with the root node as the current context and explicitly instantiated by the *xsl:apply-template* instruction. However, the former case involves some interesting subtleties for which we are not prepared at the moment.

So it's time to introduce a new instruction:

```
<xsl:apply-templates select = node-set-expression mode = qname>
  <!-- content: (xsl:sort | xsl:with-param)* -->
</xsl:apply-templates>
```

Both attributes of the *xsl:apply-template* instruction are optional. If the mode attribute is not present, the rule is applied in the default mode which is represented by *Nothing* in Haskell. If the select attribute is missing, matching is performed for all children of the current context node.

On the level of the abstract syntax we introduce a new type constructor for the instruction:

```
data Template = ...
```

```
  | TemplApply (Maybe SelectExpr)
               (Maybe ExName)         -- mode
               (Map ExName Variable)  -- passed arguments
               [SortKey]
```

The third and fourth arguments can be considered as something we just pass around at the moment. Likewise the compilation of this instruction is omitted here. Let's directly proceed to its application:

```
applyApplTempl :: Template -> Context -> [XmlTree]
applyApplTempl (TemplApply expr mode args sorting) ctx =
    applyMatchRulesToEntireContext params rules mode sortedCtx
  where
    params      = createParamSet args ctx
    sortedCtx   = applySorting sorting ctx nodes
    nodes       = maybe (getChildrenNT $ ctxGetNode ctx)
                        (flip applySelect ctx)
                        expr
    rules       = getMatchRules ctx
```

If there is no select expression we will use all child nodes of the current node to form the new context. *(A)pplySorting* will be explained in a short while. *(C)reateParamSet* instantiates the arguments which should be passed. This is necessary since the argument values are calculated by expressions and we have to pass actual values. The details will be described in the next chapter. The list of match rule candidates is fetched from the context. Or more specifically, the context refers to the entire compiled stylesheet, where these and other elements can be looked up. Then we iterate over all selected nodes using the already discussed *processContext* function.

```
applyMatchRulesToEntireContext :: ParamSet -> [MatchRule] -> Maybe ExName -> Context -> [XmlTree]
applyMatchRulesToEntireContext args rules mode ctx =
  processContext ctx (applyMatchRules args rules mode)
```

(A)pplyMatchRules iterates over all rules by a simple recursion. If there is a matching rule the result of the matching will be returned. Otherwise the XSLT default rules have to be applied. As we showed before the first match in the list will always be a best match.

```
applyMatchRules :: ParamSet -> [MatchRule] -> (Maybe ExName) -> Context -> [XmlTree]
applyMatchRules _    []           mode ctx = matchDefaultRules mode ctx
applyMatchRules args (rule:rules) mode ctx = maybe (applyMatchRules args rules mode ctx)
                                                   id
                                                   (applyMatchRule args rule mode ctx)
```

A single rule is matched when the mode of the rule is equal to the current mode and the match-expression of the rule matches the current node.

```
applyMatchRule :: ParamSet -> MatchRule -> Maybe ExName -> Context -> Maybe [XmlTree]
applyMatchRule args rule@(MatRule expr _ ruleMode _ _ _) mode ctx =
  if mode==ruleMode && applyMatch expr ctx
    then Just $ instantiateRule applyTemplate args rule $ ctxSetRule (Just rule) ctx
    else Nothing
```

When the matching is successful the current rule is set as the new context rule. This is necessary for the xsl:apply-imports instruction. *(I)nstatiateRule* is explained when we introduce variables, as it is mainly concerned with the treatment of the rules parameters. The careful reader might have noticed the surprising occurrence of *applyTemplate* as an argument to *instantiateRule*. We will elaborate on that later.

Applying imported rules

XSLT provides a mechanism to match the currents node with one of the rules which have been imported to the current context rule. This feature is similar to *super.foo()* in Java. It's useful whenever a match-rule just defines a part of the processing for a particular node and wants to forward the common processing to the rules from the imported stylesheets.

(X)sl:apply-imports does not have any arguments and is therefore trivially represented by the following data constructor:

```
data Template = ...
  | TemplApplyImports
```

The compilation is obviously not a challenge. The application is defined as follows:

```
applyImports :: Template -> Context -> [XmlTree]
applyImports (TemplApplyImports) ctx=
    applyMatchRules Map.empty rules mode ctx
  where
    rules    = getRuleImports currRule
    mode     = getRuleMode currRule
    currRule = maybe (error "apply-imports must not be called during xsl:for-each")
                     id
                     $ ctxGetRule ctx
```

(A)pplyImports retrieves the current rule from the context. The current rule must be the rule which has been selected for the current context node. Since only the stylesheet instantiation, *xsl:apply-templates* and *xsl:for-each* change the current context node, the only situation in which such a rule is not available is during the processing of a for-each instruction. For-each instructions set the context rule to *Nothing*. The imported rules and the current mode are fetched from the current rule. No parameter passing is performed in XSLT 1.0. We will pass the empty map. However, XSLT 2.0 allows the passing of parameters from apply-imports. Fortunately this would be fairly easy to implement in the future.

Applying default rules

The XSLT specification defines default rules in an XSLT-like but informal notation.

Basically, whenever no matching rule can be found the default rules apply.

If the current node is...

- ...a root or element node, matching is attempted in the current mode for all its children (including text-nodes, processing instructions and comments, but not including attributes).
- ...a text node, it is copied to the result.
- ...an attribute, its value is copied to the result.
- ...any other node, an empty result set is returned.

As a consequence, if we apply a stylesheet to an input document and absolutely no matching rules are found, the result of the transformation will consist of all text nodes of the input document[5].

5 Assuming no whitespace-stripping has been specified.

These rules can be easily stated in Haskell:

```
matchDefaultRules :: (Maybe ExName) -> Context -> [XmlTree]
matchDefaultRules mode ctx@(Ctx ctxNavNode _ _ _ _ _ stylesheet _) =

    -- rules for match="*|/"
    if isElem ctxNode
    then applyMatchRulesToChildren Map.empty rules mode ctx

    -- rule for match="text()"
    else if isText ctxNode
    then [ctxNode]

    -- rule for match="@*"
    else if isAttr ctxNode
    then [mkText $ collectTextnodes $ getChildren ctxNode]

    -- the glorious rest (PIs and comments):
    else []

  where
    rules   = getMatchRules stylesheet
    ctxNode = subtreeNT ctxNavNode
```

where:

```
applyMatchRulesToChildren :: ParamSet -> [MatchRule] -> (Maybe ExName) -> Context -> [XmlTree]
applyMatchRulesToChildren args rules mode ctx =
    applyMatchRulesToEntireContext args rules mode childCtx
  where
    childCtx = ctxSetNodes (getChildrenNT $ ctxGetNode ctx) ctx
```

(A)pplyMatchRulesToEntireContext has already been explained.

After introducing variables and parameters we will explain the application of a procedure, or named rule.

Variables and parameters

Just like general purpose programming languages XSLT offers mechanisms to define variables and parameters.

In XSLT we distinguish between:

- Global variables which can be accessed from within the entire stylesheet including all (directly or indirectly) included and imported stylesheets and all stylesheets that (directly or indirectly) include or import the stylesheet.
- Global stylesheet parameters which behave just like global variables, expect that they can receive parameter values from the outside world by some implementation defined mechanism.
- Local variables which are visible within a rule for all following instructions and all descendants of the following instructions.
- Local rule parameters which are visible for all instructions within that rule.

The following properties have to be ensured for all variables and parameters:

- Global variables and parameters shadow global variables and parameters with the same name and a lower import precedence. There must not be two global variables or parameters with the same name and the same import precedence.

- There must not be two local variables or parameters with same name visible in the same context.
- Parameters and variables share a common namespace. If no two variables with the same name are legal within a certain context, two parameters with the same name or a variable and a parameter with the same name are illegal as well.
- Local variables or parameters always shadow global variables or parameters with the same name.
- Local variables can be declared wherever an instruction can be declared.
- There is no assignment operation for variables other than initialization.

Syntactically variables and parameters are declared as follows:

```
<xsl:variable name = qname select = expression>
  <!-- content: template -->
</xsl:variable>

<xsl:param name = qname select = expression>
  <!-- Content: template -->
</xsl:param>
```

The name attribute is required. The select attribute is optional and defaults to the empty literal string. The XSLT specification allows the binding of result tree fragments (which can be constructed by the contained instructions) to variables. While this is a useful feature it is notoriously hard to implement as it requires the introducing a new data type to the XPath, which must be treated separately from the previously defined data types strings, boolean, number and node set. See appendix II.

Compilation of xsl:variable and xsl:param

Ignoring result tree fragments parameters and variables can be represented in Haskell as follows:

```
data Variable = MkVar Bool ExName Expr
```

The first argument of the *MkVar* constructor is *True* for parameters and *False* for *"normal"* variables. We define:

```
isParam :: Variable -> Bool
isParam (MkVar isP _ _) = isP
```

The compilation of a single *xsl:variable* or *xsl:param* element is defined as follows:

```
compileVariable :: XmlTree -> Variable
compileVariable node =
    MkVar modus name expr
  where
    modus   = isElemType xsltParam node
    name    = parseExName><node $ fetchAttribute node xsltName
    expr    = parseExpr><node   $ fetchAttributeWDefault node xsltSelect "''"
```

Organizing global variables and parameters

All global variables and parameters of one expanded stylesheet are compiled by:

```
compileVariables :: [XmlTree] -> Map ExName Variable
compileVariables nodes =
    foldl insertVar Map.empty $ varList
```

```
  where
    varList           = map compileVariable $ nodes
    insertVar map var = Map.insertWith (error $ "parameter or variable "++ show (getVarName var)
                                                 ++ " is already defined on this level")
                                        (getVarName var) var map
```

An error is issued in case of a name conflict. Global variables and parameters are joined with global variables and parameters from imported stylesheets in the same way named rules are joined.

```
assembleVariables :: [XmlTree] -> [(Map ExName Variable)] -> (Map ExName Variable)
assembleVariables varElems = Map.unions . (compileVariables varElems:)
```

The compiled stylesheet needs an additional field for the global variables

```
data CompiledStylesheet = CompStylesheet ... (Map ExName Variable)
```

(A)ssembleStylesheet is extended as follows:

```
assembleStylesheet :: XmlTree -> [CompiledStylesheet] -> CompiledStylesheet
assembleStylesheet xslNode imports=
    CompStylesheet ... variables
  where
    -- compiled contents:
    variables             = assembleVariables varElems importedVariables
    ...

    -- element content:
    (varElems, rest2)     = partition (\node -> isElemType xsltVariable node
                                            || isElemType xsltParam    node) rest
    ...

    -- imported stuff:
    importedVariables     = map getVariables revImports
    ...
```

Initializing globals

Before we proceed to the concrete implementation, we should consider an algorithmic problem first. Global variables and parameter can depend on each other in no particular order. For example variable *a* depends on the value of variable *c* which depends on the value of variable *b*. There are no limitations on these dependencies, as long as no cycles are created.

It is instructive to sketch this problem with a small Haskell program. To make it more tractable we use the following poor man's XPath expressions:

```
data Expr = MkLit String | MkVarRef String
```

An expression can be either a literal text or a variable reference. For the evaluation of an expression a context consisting of variable name to variable value bindings is needed:

```
computeVal :: [(String, String)] -> Expr -> String
computeVal _    (MkLit lit)     = lit
computeVal vars (MkVarRef name) = fromJust $ lookup name vars
```

We can now define such an environment of global variables in a recursive fashion:

```
variables :: [(String, String)]
variables = [ ("a", computeVal variables (MkVarRef "c"))
            , ("b", computeVal variables (MkLit "Fred"))
```

```
          , ("c", computeVal variables (MkVarRef "b")) ]
```

The trick is that all variables are already a part of the environment in which they are evaluated. It wouldn't work if the environment was evaluated strictly. But with lazy-evaluation we resolve the dependencies between the variables practically for free.

```
> variables [enter]
"[(\"a\",\"Fred\"),(\"b\",\"Fred\"),(\"c\",\"Fred\")]"
```

If we replace *MkLit "Fred"* with *MkVarRef "a"* we obtain:

```
> variables [enter]
"[(\"a\",\"*** Exception: <<loop>>
```

It is possible for the runtime environment to detect the infinite recursion at this point. Basically the term *computeVal variables MkVarRef "c"* can be marked as just being evaluated. When we attempt to evaluate it for the second time during the evaluation of *computeVal variables MkVarRef "a"* it is clear that *computeVal variables MkVarRef "c"* is just being evaluated and is not in head normal form. That is, it has not passed the first (outermost) evaluation step. Therefore an infinite recursion must have occurred. Unfortunately this does not work in the general case. For example:

```
<xsl:variable name="a" select="concat('x', $a)" />
```

can lead to an infinite recursion, as it is always possible to evaluate one more letter of *a*, but never all of its letters. Let us consider this as good enough for now. Our variables are lazily evaluated just like normal Haskell variables and share the same benefits and limitations. It is not advisable to take advantage of this *"feature"*, if you want to write portable stylesheets. However, it could be interesting to play around with it a bit more to see whether it would allow interesting idioms in XSLT.

The original stylesheet application is defined below:

```
applyStylesheetWParams :: Map ExName Expr -> CompiledStylesheet -> XmlTree -> [XmlTree]
applyStylesheetWParams inputParams cs@(CompStylesheet matchRules _ vars _ strips _) rawDoc =
    applyMatchRules Map.empty matchRules Nothing ctxRoot
  where
    ctxRoot   = Ctx docNode [docNode] 1 1 gloVars Map.empty cs Nothing
    gloVars   = Map.map (evalVariableWParamSet extParams ctxRoot) vars
    extParams = Map.map (flip evalXPathExpr ctxRoot) inputParams
    docNode   = ntree $ stripDocument strips rawDoc
```

Global parameters are initialized by an implementation specific mechanism. We choose to allow the user to pass a number of parameter name to XPath expression mappings. These are evaluated to parameter name to XPath value mappings. Likewise, the global variables- and parameters are evaluated to variable-name to XPath value mappings. Both the passed parameters and the actual parameters and variables are evaluated recursively in the same context in which they can already be used. As a result it is possible to pass parameter expressions like *($i + $j)*3* which refer to other variables and parameters. If a passed parameter does not refer to any of the global stylesheet parameters it will be ignored just like passed parameters which do not refer to rule parameters must be ignored. As a result of Haskell's laziness, such parameters will never be evaluated. Whitespace-stripping for the input document will be explained on page 59. *Map.empty* in the context creation refers the empty set of local parameters, *Nothing* to the unspecified current rule.

For convenience we supply the user with a parameterless version of *applyStylesheet*:

```
applyStylesheet :: CompiledStylesheet -> XmlTree -> [XmlTree]
applyStylesheet  = applyStylesheetWParams Map.empty
```

A single variable is evaluated to an XPath value as follows:

```
evalVariableWParamSet :: ParamSet -> Context -> Variable -> XPathValue
evalVariableWParamSet ps ctx (MkVar isParam name exprVar) =
    if isParam
    then maybe resultFromVar id $ Map.lookup name ps
    else resultFromVar
  where
    resultFromVar = evalXPathExpr exprVar ctx
```

This function captures the only difference between variables and parameters. A parameter tries to look up the its current value from the set of passed parameters and defaults to its own value if no parameter value with the right name is passed. A variable is directly evaluated regardless of any passed parameter with the same name.

Treating local variables

In principle local variables can be declared wherever an instruction is permitted. We define a new instruction constructor for local variables.

```
data Template = ...
  | TemplVariable Variable
```

The compilation is trivial:

```
compileTemplVariable :: XmlTree -> Template
compileTemplVariable = TemplVariable . compileVariable
```

XSLT does not prohibit variables which are used as standalone instructions like in:

```
<xsl:for-each select="*">
  <xsl:variable name="name" select="name()" />
</xsl:for-each>
```

However, such variables do not have any conceivable meaning. They are suspicious enough to issue a warning:

```
applyTemplate :: Template -> Context -> [XmlTree]
...
applyTemplate t@(TemplVariable v) =
  trace ("Warning: Unreacheable variable: " ++ show (getVarName v)) const []
```

In the normal case variables will always occur within a composite instruction. We must enhance the application of composite instructions to cope with local variables.

```
applyComposite :: Template -> Context -> [XmlTree]
applyComposite (TemplComposite templates) ctx =
    fst $ foldl applyElem ([], ctx) templates
  where
    applyElem :: ([XmlTree], Context) -> Template -> ([XmlTree], Context)
    applyElem (nodes, ctx) (TemplVariable v) = (nodes, processLocalVariable v Map.empty ctx)
    applyElem (nodes, ctx) t                 = (nodes ++ applyTemplate t ctx, ctx)
```

All *normal* elements of a composite instruction will add new nodes to the result tree and leave the context unchanged. Local variables do not create result nodes. They add a new variable name to XPath Value binding to the context for all following instruction.

While our implementation might be reasonably clear it is not reasonably efficient. We accumulate a list by constantly appending on the left side. As you might recall the runtime for *(++)* depends linearly on the size of its first argument. Therefore, the entire foldl has a quadratic complexity. We can do better than that:

```
applyComposite :: Template -> Context -> [XmlTree]
applyComposite (TemplComposite templates) ctx =
    concat $ reverse $ fst $ foldl applyElem ([], ctx) templates
  where
    applyElem :: ([[XmlTree]], Context) -> Template -> ([[XmlTree]], Context)
    applyElem (nodes, ctx) (TemplVariable v) = (nodes, processLocalVariable v Map.empty ctx)
    applyElem (nodes, ctx) t                 = (applyTemplate t ctx : nodes, ctx)
```

Now we accumulate the result tree fragments in a list. Reverse once. Concat once. Linear complexity! That's all.

Processing a local variable simply means evaluating a variable and adding the new variable name to XPath value binding to the context. Unlike global variables local variables are not visible in the context in which they are introduced.

```
processLocalVariable :: Variable -> ParamSet -> Context -> Context
processLocalVariable var@(MkVar _ name _) arguments ctx =
    addVariableBinding name val ctx
  where
    val = evalVariableWParamSet arguments ctx var
```

Passing local and global variables to XPath

Within XSLT we can introduce new variable bindings from qualified names to XPath-values; however, all variable references occur within XPath expressions. Therefore we need to pass all local and global variables to XPath. We can revise our XPath evaluation function from page 11:

```
evalXPathExpr :: Expr -> Context -> XPathValue
evalXPathExpr expr ctx@(Ctx node nodelist pos len globVars locVars _ _) =
    filterXPath $ evalExpr (vars,[]) (pos, len, node) expr (XPVNode nodelist)
  where
    filterXPath (XPVError err)    = error err
    filterXPath xpv               = xpv
    vars                          = map (\(name, val) -> ((namePrefix name, localPart name), val))
                                        varList
    varList                       = Map.toAscList $ locVars `Map.union` globVars
```

First we create a left-biased union of the local and global variables. That means local variables always shadow global variables with the same name. Then we have to convert our structure to an associative list which is expected by XPath. Some performance concerns are justified at this point. The union operation is reasonably efficient due to lazy evaluation. That is we do not need to create a full union of all elements just to retrieve a single element. The conversion to an associative list means; however, that the union must be evaluated for all variables preceding the variable name which is looked up. It could be reasonable to change the XPath implementation to use *proper* maps instead of associative lists. Nonetheless, if no variable is accessed by an expression (which is most often the case), nothing has to be done for the variables. This is one more example where Haskell's lazy evaluation lets us *get away* with something that would be prohibitive in a language with eager evaluation.

Passing parameters to procedures

We have deferred the discussion of the application of procedures until now. Both named rules and procedures can have parameters. But when we discussed the application of a match rule the emphasis was on the selection of the right rule while in the case of the application of a procedure the emphasis is on the correct passing of parameters.

Consider the following simple recursive XSLT-procedure:

```
<xsl:template name="procedure">
  <xsl:param name="depth" select="3"/>
  <xsl:element name="result_{$depth}">
    <xsl:if test="$depth &gt; 0">
      <xsl:call-template name="procedure">
        <xsl:with-param name="depth" select="$depth - 1"/>
      </xsl:call-template>
    </xsl:if>
  </xsl:element>
</xsl:template>
```

If we call this procedure without any arguments:

```
<xsl:call-template name="procedure" />
```

we obtain:

```
<result_3><result_2><result_1><result_0/></result_1></result_2></result_3>
```

Two things should be noted:

- There is no name conflict between the local variable (or parameter) depth and the passed argument with the same name.
- The default-value (three) for the parameter depth will only be used if no parameter with that name is passed.

We can define the following constructor for xsl:call-template:

```
data Template = ...
  | TemplCall ExName (Map ExName Variable)
```

The passed arguments (or *xsl:with-param* elements) are compiled to normal variables.

```
compileCallTempl :: XmlTree -> Template
compileCallTempl node =
    TemplCall name args
  where
    name = parseExName><node $ fetchAttribute node xsltName
    args = compileVariables  $ filter (isElemType xsltWithParam) $ getChildren node
```

In order to apply a procedure we must:

- look up the procedure name in the environment.
- evaluate the list of passed parameters.
- instantiate the procedure with the evaluated parameters.

```
applyCallTempl  :: Template -> Context -> [XmlTree]
applyCallTempl (TemplCall name args) ctx =
    instantiateRule applyTemplate params rule ctx
  where
    params      = createParamSet args ctx
    rule        = maybe errNoRule id $ Map.lookup name rules
    rules       = getNamedRules $ ctxGetStylesheet ctx
    errNoRule   = error $ "No rule with qualified name: " ++ show name
```

If no procedure with the right name is found an error is issued. If a procedure is found it will always be the one with the highest import precedence, as we showed. Creating the parameter set means evaluating the variable name to XPath expression bindings to variable name to XPath value bindings.

```
createParamSet :: Map ExName Variable -> Context -> ParamSet
createParamSet wParamList ctx = Map.map (evalVariableWParamSet Map.empty ctx) wParamList
```

The names of passed parameters are not bound to the environment. One passed parameter cannot refer to the value of another passed parameter. It is not an error if a passed parameter has the same name as a local variable or parameter.

The instantiation of a named rule (procedure) or a match rule is done by the polymorphic function *instantiateRule*:

```
instantiateRule :: Rule a =>
    (Template -> Context -> [XmlTree]) -> ParamSet -> a -> Context -> [XmlTree]
instantiateRule applyTemplate args rule ctx =
    applyTemplate (getRuleContent rule) ctxNew
  where
    ctxNew = processParameters (getRuleParams rule) args $ clearLocalVariables ctx
```

Whenever a rule is instantiated all local variables and parameters are cleared from the environment. Surprisingly, the *applyTemplate* function must be passed as the first argument to *instantiateRule*. This way we split a *mutual recursive group* which would otherwise hinder the type checker from accepting the code. This issue is described in detail in Mark P. Jones' paper *Typing Haskell in Haskell* chapter 11.6.3 *combined binding groups*. The general opinion seems to be that Haskell's current typing rules are a bit too restrictive at this point and should be relaxed in the future.

The parameters are evaluated in the order of their declaration. A parameter cannot refer to itself when it is evaluated; however, any parameter can refer to all parameters which have been declared before it:

```
processParameters :: [Variable] -> ParamSet -> Context -> Context
processParameters params arguments ctx =
  foldl (\c v -> processLocalVariable v arguments c) ctx params
```

Another quick look back

We will keep with our tradition of occasionally holding on for a second, taking a step back and look where we are. When we took our first look back we realized that we could already process many useful transformations, although we had just implemented a tiny fraction of the XSLT language. Now there is more to look at. We cannot just process many useful transformations. We are already able to perform any computable transformation from a source- to a result XML document[6] within the subset of XSLT that we have implemented by now. We have already encountered a small recursive procedure. We can pull arbitrary values from an input document and with the basic string functions from the XPath library,

6 Excluding those properties of XML-documents which cannot be observed by XSLT: DTDs and entity references.

combined with recursive procedures, conditional processing and parameter passing, we can compute arbitrary strings. These arbitrary strings can be used to create elements, attributes, text nodes, comments and processing instructions.

Therefore, at least theoretically, any feature which is missing by now can be implemented in our XSLT subset. While most of the times our algorithmic tasks are much better solved with Haskell than it would be possible with XSLT, there are some situations, like whitespace stripping on stylesheets, in which an XSLT implementation deserves consideration. By no means shall that imply that XSLT is *fit* for any kind of XML transformation, nor that it is designed to be that. While XSLT has been used to simulate a Universal Turing Machine[7], it seems that there is no XSLT processor yet which is implemented entirely in XSLT.

Sorting

By default *xsl:for-each* and *xsl:apply-templates* process the selected nodes in document order. XSLT users can optionally supply both instruction with a list of sorting criteria. For example, we can modify the original example from the introduction to sort the items of the invoice by descending price (the most expensive items come first) and alphabetically (items with the same price are ordered alphabetically):

```
<html xsl:version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <body>
    <table>
       <xsl:for-each select="*/item">
         <xsl:sort select="@value" data-type="number" order="descending"/>
         <xsl:sort select="@name"/>
         <tr>
           <td><xsl:value-of select="@name"/></td>
           <td><xsl:value-of select="@value"/></td>
         </tr>
       </xsl:for-each>
    </table>
  </body>
</html>
```

Any *xsl:sort* element can have a select attribute with the default value *"."*, a data-type attribute which evaluates to either *number* or *<u>text</u>* (<u>default</u>) and an order attribute which evaluates to either *descending* or *<u>ascending</u>* (<u>default</u>). There is also a *case-order* attribute which we will ignore here as its semantics are mostly implementation defined. Our *"implementation defined"* choice is to simply ignore it. We will define a case insensitive lexicographical comparison.

An *xsl:sort* element can be represented by:

```
data SortKey = SortK StringExpr -- select
                     StringExpr -- data-type: number or text(default)
                     StringExpr -- order: ascending(default) or descending
```

The compilation of a *SortKey* is defined as:

```
compileSortKey :: XmlTree -> SortKey
compileSortKey node =
    SortK expr dataType order
  where
    expr     = parseStringExpr><node $ fetchAttributeWDefault node xsltSelect "."
```

7 Bob Lyons implemented one which can be found at: http://www.unidex.com/turing/utm.htm

```
    dataType = parseAVT><node        $ fetchAttributeWDefault node xsltDataType "text"
    order    = parseAVT><node        $ fetchAttributeWDefault node xsltOrder "ascending"
```

The XSLT specification defines the *data type* and *order* attribute values as attribute value templates. That allows us to provide *ascending* or *descending* as stylesheet parameters or to compute them in some other way.

The result of the select expression is always interpreted as a string, but it can be optionally converted to a number afterwards to ensure that numeric values are compared numerically. In the example above we don't want to sort the values of the items on the invoice lexicographically that is *"100" < "60" < "7000" < "9"*. We want to sort them numerically that is *9 < 60 < 100 < 7000*.

If we want to implement sorting efficiently we have to decide at which point each of the expressions must be evaluated:

The XSLT specification does not define in which context the *data-type* and order attribute value templates must be evaluated. However, the only reasonable choice is to evaluate them exactly once in the context in which *xsl:apply-templates* or *xsl:for-each* have been instantiated, since we do not want to have different values for each node which has to be sorted. On the other hand the select expression must be evaluated once for each node.

It makes sense to perform string to number conversions only once, when the select expression is evaluated. We define the following data-type for the result of a select expression:

```
type SortVal = Either Float String
```

The application of an individual *xsl:sort* element is performed once before the actual sorting takes place. Applying an *xsl:sort* element produces two results:

- An extract function which extracts a *SortVal* from the current context. This function will be applied once for every node which has to be sorted.
- A comparison-function which performs a comparison between two sort-values. This function will be applied at most once for each comparison in the sorting procedure. Our algorithm will perform at most *n log n* comparisons where *n* is the number of nodes.

Both functions are generated by the following procedure:

```
applySortKey :: SortKey -> Context -> ( Context -> SortVal
                                      , SortVal -> SortVal -> Ordering)
applySortKey (SortK expr typeATV orderATV) ctx =

    if typ/="number" && typ/="text"
    then error $ "unsupported type in xsl:sort: " ++ typ

    else if ord/="ascending" && ord/="descending"
    then error $ "order in xsl:sort element must be ascending or descending. Found: " ++ ord

    else (extractFct, cmpFct)

  where

    isNumber       = typ == "number"
    isDesc         = ord == "descending"
    ord            = applyStringExpr orderATV ctx
    typ            = applyStringExpr typeATV ctx

    extractFct ctx = let val = applyStringExpr expr ctx in
                       if isNumber
                         then Left $ readWDefault (-1.0 / 0.0) val
                         else Right val

    cmpFct a       = (if isDesc then invertOrd else id)
                   . if isNumber then cmpNumber a else cmpString a
    cmpNumber (Left n1)  (Left n2)  = compare n1 n2
    cmpString (Right s1) (Right s2) = compare (map toLower s1) (map toLower s2)
```

Sanity checking is done for the *order* and *data-type* attributes. The extracting function defaults to minus infinity whenever the string value of the current node cannot be interpreted as a number. The comparison function is the concatenation of an optional inversion-function for descending order and a data-type specific comparison. The inversion function for an ordering is defined as follows:

```
invertOrd :: Ordering -> Ordering
invertOrd EQ = EQ
invertOrd LT = GT
invertOrd GT = LT
```

The number comparison is trivial and the string comparison function implements a case-insensitive ordering by simply converting both of it's attributes to the lower case. While our implementation of the string comparison is not strictly non-conforming, the W3C suggests an implementation based on the Unicode Collation Algorithm[8].

The basic idea of this algorithm is to use at least three different levels of collation, where each level takes precedence over the following one. For example:

```
Københaven < København   based on level 1 (here: spelling haven<havn)
Kobenhavn  < København   based on level 2 (here: umlauts o<ø)
københavn  < København   based on level 3 (here: lower-/upper-case k<K)
```

Since level 1 takes precedence over level 2 and level 2 takes precedence over level 3:

```
Københaven < Kobenhavn   level 1 (haven<havn) over level 2 (ø>o)
Kobenhavn  < københavn   level 2 (o<ø) over level 3 (K>k)
```

This example was arbitrarily chosen. Level 2 does not necessarily deal with umlauts and level 3 does not have to deal with lower or upper case. The meaning is language dependent.

It seems that there is no Haskell library today which implements the Unicode Collation Algorithm and an implementation of it is far beyond the scope of this work. It could easily constitute a master thesis on its own. These subtleties of text ordering should explain our reluctance to give a meaning to the case-order attribute. A naive implementation would likely do more harm than good, as it would implicitly work on the wrong level. Therefore we choose case insensitive ordering. Our implementation will produce meaningful results for English, but not e.g., for German. In the case of German it would be reasonable to transform umlauts to the corresponding non-umlauts before comparing: ä to a, ö to o, ü to u and ß to s. Other languages require different conversions.

Now that we have an extraction and a comparison function for each sorting criterion, we must combine them to sort a nodeset according to all these criteria. The XSLT

8 Unicode Technical Standard #10: http://www.unicode.org/unicode/reports/tr10/index.html

specification requires a stable sorting, that is all nodes that are equal according to all criteria must remain in document order.

There are two principal algorithms to perform a stable sorting according to many criteria. The first is to perform a stable sort of the entire list according to the least significant criterion first. Then sort it again according to the second least significant criterion and so on until all criteria are used. While this algorithm is simple and easy to implement, it is not particularly efficient, since many potentially unnecessary comparisons are made.

We choose the second and more conventional algorithm. We combine all sorting criteria to a single combined criterion. That means we combine the extracting functions to one function which extracts a list of SortVal(s) from a node. Likewise we combine the comparison functions to one function which compares two lists of SortVal(s).

```
applySorting :: [SortKey] -> Context -> [NavXmlTree] -> Context
applySorting [] ctx nodes = ctxSetNodes nodes ctx
applySorting sortKeys ctx nodes =
    ctxSetNodes resultOrder ctx
  where
    resultOrder          = snd $ unzip sortedKVs
    sortedKVs            = sortBy compKV keysWVals
    keysWVals            = zip keys nodes
    keys                 = map extract nodes
    (extrFs, cmpFs)      = unzip $ map (flip applySortKey ctx) sortKeys

    -- helper functions:
    extract node         = map ($ ctxSetNodes [node] ctx) extrFs
    compKV (k1,_) (k2,_) = compressOrds $ compares k1 k2
    compares             = zipWith3 (($) $) cmpFs
    compressOrds         = maybe EQ id . find (/=EQ)
```

Each list of extracted SortVal(s) is zipped together with the corresponding node. The comparison function *compares* combines the list of individual comparison functions for each criterion to a compound function which produces a list of comparison results. The *zipWith3 (($) $)* expression demonstrates a typical higher order use of the *($)*-operator, as the second arguments of zipWith3 is a list of functions, which must be applied to lists of values. *(C)ompressOrds* compresses that list of single results to one result which will be equal when all individual comparisons were equal. Otherwise, the first non-equal comparison result is the result of the combined comparison. In a language with eager evaluation this approach would defeat our purpose, since we were trying to reduce the number of comparisons. However, as a result of lazy evaluation only the those comparisons must be evaluated, which are necessary for compressOrds to yield a result. This programming style allows us to define data-structures for intermediate results which are potentially very expensive to compute. Since these structures will not be fully evaluated most of the time, no large runtime overhead occurs. This kind of programming is more declarative than it would be possible in a language with eager evaluation. In chapter 17.4. *Data-directed programming* of his book *Haskell, The Craft of Functional Programming* Simon Thompson demonstrates several other intriguing applications of this programming style.

Attribute sets

Literal result elements, *xsl:copy* and *xsl:element* can be supplied with a list of so called attribute set names. For example the following stylesheet adds a *processed="TRUE"* attribute to each element of an input document.

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/|*">
    <xsl:copy use-attribute-sets="as">
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
  <xsl:attribute-set name="as">
    <xsl:attribute name="processed">TRUE</xsl:attribute>
  </xsl:attribute-set>
</xsl:transform>
```

Attribute sets may only contain xsl:attribute instructions and can have use-attribute-sets attributes themselves. The attributes from the attribute sets are instantiated before the content of the *xsl:copy*, *xsl:element*, *xsl:attribute-set* and literal result elements. Therefore any attribute added explicitly within the content of such an instruction takes precedence over attributes injected from attribute sets. In case of literal result elements, the attributes injected from attribute sets take precedence over attributes which are specified on the literal result element itself.

(X)sl:attribute-set is represented by:

```
data AttributeSet = AttribSet ExName UsedAttribSets Template
```

where:

```
newtype UsedAttribSets = UsedAttribSets [ExName]
```

Using an arbitrary template as the attribute set's content is certainly general enough to deal with a list of xsl:attribute instructions. We can restrict the usage when compiling the *xsl:attribute-set* element. The restriction to include only *xsl:attribute* elements is not needed by our implementation and general templates would allow meaningful applications like conditionally adding an attribute with *xsl:if*. Non-xsl-attribute content is simply filtered out during the compilation. More sophisticated error handling could be added in the future:

```
compileAttributeSet :: XmlTree -> AttributeSet
compileAttributeSet node =
    AttribSet name usedsets template
  where
    name     = parseExName><node $ fetchAttribute node xsltName
    usedsets = UsedAttribSets    $ parseExNames><node
                                   $ fetchAttributeWDefault node xsltUseAttributeSets ""
    template = compileTemplate   $ filter (isElemType xsltAttribute) $ getChildren node
```

The XSLT specification defines intriguing rules on how attribute sets from different import files must be merged in case of a name conflict. All attribute sets must be instantiated and attributes created from attribute sets with higher import precedence override those created from attribute sets with a lower import precedence. We implement this by providing a list of attribute sets for each attribute set name. This list is ordered by ascending import priority. The attribute set with the lowest priority will add its attributes first, these can be overridden by all subsequent attribute sets with a higher import precedence.

```
assembleAttrSets :: [XmlTree] -> [Map ExName [AttributeSet]] -> Map ExName [AttributeSet]
assembleAttrSets attsetElems =
    foldr (Map.unionWith (++)) localAttribSets
  where
    localAttribSets                = foldr insertAs Map.empty $ map compileAttributeSet attsetElems
    insertAs a@(AttribSet nam _ _) = Map.insertWith (++) nam [a]
```

Even within one stylesheet, or one stylesheet and all its includes, there can be more than one attribute set with the same name. In that case we order the attribute sets in document order. The attribute sets are added to the compiled stylesheet:

```
data CompiledStylesheet = CompStylesheet ... (Map ExName [AttributeSet])

getAttributeSets :: CompiledStylesheet -> Map ExName [AttributeSet]
```

(A)ssembleStylesheet is extended as follows:

```
assembleStylesheet :: XmlTree -> [CompiledStylesheet] -> CompiledStylesheet
assembleStylesheet xslNode imports=
    CompStylesheet ... attsets
  where
    -- compiled contents:
    attsets               = assembleAttrSets attsetElems importedAttribSets
    ...

    -- element content:
    (attsetElems, rest3)  = partition (isElemType xsltAttributeSet) rest2
    ...

    -- imported stuff:
    importedAttribSets    = map getAttributeSets imports
    ...
```

We can revisit our *applyElement* function from page 14 to deal with attribute sets. The added code is marked by bold letters.

```
applyElement :: Template -> Context -> [XmlTree]
applyElement (TemplElement compQName uris attribSets template) ctx =
    return $ createElement name uris fullcontent
  where
    name        = applyComputedQName compQName ctx
    fullcontent = applyAttribSets [] attribSets ctx ++ applyTemplate template ctx
```

We have silently added another attribute to the *TemplElement* constructor. It is the cursively written namespace prefix to namespace URI mapping effective for the instruction. The *applyCopy* function is modified in the same fashion. The first argument of the *applyAttribSets* function is an empty call stack. This call stack is necessary to detect a cyclic use of xsl:attribute which is forbidden by the XSLT specification. *(A)pplyAttribSets* instantiate each used attribute set and concatenates the result.

```
applyAttribSets :: [ExName] -> UsedAttribSets -> Context -> [XmlTree]
applyAttribSets callstack (UsedAttribSets sets) ctx =
  concatMap (\name -> applyAllAttrSetForName callstack name ctx) sets
```

(A)pplyAllAttrSetForName must look up all attribute sets which are bound to the current name and instantiate them in order:

```
applyAllAttrSetForName :: [ExName] -> ExName -> Context -> [XmlTree]
applyAllAttrSetForName callstack name ctx =

    if name `elem` callstack
    then error $ "Attribute-Set " ++ show name ++ " is recursively used." ++
                 concatMap (("\n  used in "++) . show) callstack

    else if isNothing attrset
    then error $ "No attribute set with name: " ++ show name

    else concatMap (flip (applyAttribSet (name:callstack)) ctx) $ fromJust attrset

  where
```

```
    attrset = Map.lookup name $ getAttributeSets $ ctxGetStylesheet ctx
```

Sanity checking for the recursive use of an attribute set is performed. Just like in the case of recursive includes a nicely formatted error message is presented in case of a recursion:

```
Attribute-Set a2 is recursively used.
  used in a4
  used in a3
  used in a2
  used in a1
```

It is also an error if no attribute set is bound to the current name. Otherwise all attribute sets are instantiated in order of ascending import precedence. A single attribute set is instantiated by first instantiating all its used attribute sets (, that's where the potential recursion comes into play) and then its content:

```
applyAttribSet :: [ExName] -> AttributeSet -> Context -> [XmlTree]
applyAttribSet callstack (AttribSet _ usedSets content) ctx =
    applyAttribSets callstack usedSets ctx ++ applyTemplate content ctx
```

The way attribute sets are specified is a showcase for unnecessarily complicated and inconsistent semantics in the XSLT specification. Imagine the rules required that only the attribute set with the highest import precedence should be instantiated and would not allow attribute sets to use further attribute sets. In this case we could have simply regarded attribute sets as a convenient notation for named rules and could have implemented them as a derived form. The semantics would be easier to explain and it is hardly conceivable that users actually require the intriguing rules for merging of attribute sets.

Whitespace stripping

Stylesheets and input documents are *stripped* in XSLT according to two somewhat different sets of rules. The following rules apply to both:

- XSLT will only strip text nodes that consist of whitespace characters alone. Heading, trailing and in-between whitespace characters in none-empty text nodes will always be preserved.
- Whitespace stripping takes place before the actual processing.
- By default, all whitespace nodes of a stylesheet are stripped.
- By default, all whitespace nodes of an input document are preserved.
- Whitespace stripping for both input documents and stylesheets can be parametrized by the user; however, the mechanisms in both cases are totally different.

The general whitespace stripping rules can be implemented as a higher order operation:

```
stripSpaces :: (Bool -> XNode -> Bool) -> Bool -> XmlTree -> XmlTree
stripSpaces f def =
    fromJustErr "stripSpaces (internal error)" . filterTreeCtx step def
  where
    step strip node
     | isElem node           = (f strip node, True)
     | isWhitespaceNode node = (strip       , not strip)
     | otherwise             = (strip       , True)
```

We pass a function which determines whether the children of an element node should be stripped. This function has access to the previous strip status and the element node itself. The default strip status is passed as the second argument. It is *True* for stylesheets and *False* for input documents. *(S)tripSaces* itself is based on the higher order function *filterTreeCtx* which is defined as follows:

```
filterTreeCtx :: Tree t => (c -> a -> (c, Bool)) -> c -> t a -> Maybe (t a)
filterTreeCtx p c tree =
  if b
    then Just $ mkTree node $ mapMaybe (filterTreeCtx p cN) $ getChildren tree
    else Nothing
  where
    (cN, b) = p c node
    node    = getNode tree
```

A whitespace node is identified by the following procedure:

```
isWhitespaceNode :: (XmlNode n) => n -> Bool
isWhitespaceNode = maybe False (all isSpace) . getText
```

Whitespace stripping on stylesheets

Whitespace stripping on stylesheets is performed according to the following rules:

- Whitespace nodes below the *xsl:text* element are always preserved.
- Whitespace nodes which have an ancestor node with an *xml:space* attribute with the value *preserve* and no closer ancestor node with an *xml:space* attribute with the value *default* are always preserved.
- All other whitespace nodes are stripped from the stylesheet.
- *(X)ml:space attributes are not stripped from literal result elements.*

These stylesheet-specific whitespace stripping rules are captured below:

```
stripStylesheet :: XmlTree -> XmlTree
stripStylesheet =
    stripSpaces isStrip True
  where
    isStrip strip' node =
      not (isElemType xsltText node)
      && (maybe strip' (=="default") $ tryFetchAttribute node xmlSpace)
```

Alternatively, we could have implemented the whitespace stripping for stylesheets in XSLT itself. This approach is captured in appendix III on page 73.

Whitespace stripping on input documents

According to the general rules, whitespace nodes in input documents are generally preserved. Imagine we have an input document of which we want to strip most whitespace nodes; however, we are aware that text nodes within elements of the *text* namespace (, *just an example, really...*) demand careful treatment of whitespace nodes. This can be achieved by the following stylesheet:

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                             xmlns:text="http://text.org">

<xsl:strip-space elements="*" />
<xsl:preserve-space elements="text:*" />

<xsl:template match="/">
  <xsl:copy-of select="/" />
```

```
</xsl:template>

</xsl:transform>
```

Both *xsl:strip-space* and *xsl:preserve-space* have an element attribute which contains a whitespace separated list of name tests. We have already explained name-tests on page 36. As a short reminder a name test is one of: *name, ns:name, ns:** or *. Marking an element by *xsl:presereve-space* is generally only meaningful if there is more general name-test in *xsl:strip-space*, as in the example above.

The rules for whitespace stripping on input documents are:

- Whitespace nodes are stripped from an element, if it is matched by a name test of an *xsl:strip-space* element and it is not matched by a name-test for an *xsl:preserve-space* element with a higher import precedence or the same input precedence and a higher default priority.
- There must not be two name tests in *xsl:strip-space* or *xsl:preserve-space* that match an element with the same import precedence and default priority. This is an error even if both of these matches are within either *xsl:strip-space* or *xsl:preserve-space*.

We can implement the lookup for the strip-, or preserve space status of an element from the input document in terms of the general matching procedure, which we described on page 36. However, we can take advantage of the limitations here and devise a more efficient special purpose matching procedure.

A name test can be simply represented by an expanded name.

```
type NTest = ExName
```

All strip- or preserve space name tests of one expanded stylesheet can be represented by a mapping from *NTest* to *Bool*, where *True* indicates strip and *False* indicates preserve:

```
type Strips = Map NTest Bool
```

We need one such mapping for each import precedence. We can enhance the compiled stylesheet datatype by a list of these maps ordered by descending import precedence:

```
data CompiledStylesheet = CompStylesheet ... [Strips]

getStrips :: CompiledStylesheet -> [Strips]
```

The lookup within one such map of markers of a single import precedence is defined below:

```
lookupStrip1 :: ExName -> Strips -> Maybe Bool
lookupStrip1 name spec =
    if      isJust nameMatch then nameMatch
    else if isJust prefMatch then prefMatch
    else if isJust globMatch then globMatch
    else Nothing
  where
    nameMatch = Map.lookup (       name             ) spec
    prefMatch = Map.lookup (ExName "*"  $ exUri name) spec
    globMatch = Map.lookup (ExName "*"  ""          ) spec
```

The idea is to perform the lookup with descending priority. We remember that * has the priority *-0.5*, *ns:** has the priority *-0.25* and *name* and *ns:name* have the priority *0.0*. Therefore, we first try to find a name test for the exact expanded name. Secondly, we try to find a nametest for *ns:**. Then we must test for *. The second and third case coincide in case *name* has a null namespace URI, which is represented by the empty string. In case of a non-match *Nothing* is returned, *Just True* indicates strip, *Just False* indicates preserve. The entire lookup procedure is defined as:

```
lookupStrip :: ExName -> [Strips] -> Bool
lookupStrip name = head . (++[False]) . mapMaybe (lookupStrip1 name)
```

This function makes heavy use of lazy evaluation again. We construct a list of all possible matches ordered by descending import precedence. Then we append the default value to the list, which ensures that the list is never empty and that the correct value (*preserve*) will be delivered in case no match is found. Then we take the head of this list and ignore the rest.

The entire whitespace stripping function is listed below:

```
stripDocument :: [Strips] -> XmlTree -> XmlTree
stripDocument strips =
  stripSpaces (\_ n -> lookupStrip (mkExName $ fromJust $ getElemName n) strips) False
```

Ignoring the current strip-status *(_)* reflects the non-transitivity of *strip-space* or *preserve-space* markers.

Namespace aliasing

In order to create XSLT documents from XSLT documents, we need some mechanism to disambiguate the XSLT elements and attributes we want to create from those we use to create them. With our current implementation we could do this by using *xsl:element* and *xsl:attribute* for any XSLT attribute we intent to create. However this becomes unwieldy rather quickly. Therefore XSLT has a namespace aliasing mechanism, which is demonstrated below:

```
<xsl:transform version    = "1.0"
               xmlns:xsl  = "http://www.w3.org/1999/XSL/Transform"
               xmlns:axsl = "http://xsldude.org" >

  <xsl:namespace-alias stylesheet-prefix="axsl" result-prefix="xsl" />

  <xsl:template match="/" >
    <lre-stylesheet axsl:version="1.0" />
  </xsl:template>

</xsl:transform>
```

The result of this stylesheet is (, for any input document):

```
<lre-stylesheet xmlns:axsl="http://www.w3.org/1999/XSL/Transform" axsl:version="1.0"/>
```

The *axsl* prefix refers now to the XSLT namespace. It might seem surprising that we do not use the *xsl* prefix, but this approach avoids name clashes. The XSLT specification imposes no restrictions on generated prefixes. We also stayed faithful to our premise: *When in doubt do it as XALAN does*.

```
type NSAliasing = Map String String
```

All namespace aliases of the entire stylesheet with all its imports can be stored in a single map in the same way we managed to store all procedures in one map.

```
assembleAliases :: [XmlTree] -> [NSAliasing] -> NSAliasing
assembleAliases nsAliasElems =
    Map.unions . (localAliases:)
  where
    localAliases          = foldr addAlias' Map.empty nsAliasElems
    addAlias' node        = uncurry (addAlias><node) $ compileAlias node
```

The compilation of a single *xsl:namespace-alias* element just creates a tuple of the old and new prefix:

```
compileAlias :: XmlTree -> (String, String)
compileAlias node =
  (fetchAttribute node xsltStylesheetPrefix, fetchAttribute node xsltResultPrefix)
```

These are transformed to their respective URIs by the *addAlias* function. Duplicate definitions on this level are an error.

```
addAlias :: UriMapping -> String -> String -> NSAliasing -> NSAliasing
addAlias uris oldPr newPr =
    Map.insertWith (error $ "duplicate mapping for " ++ old) old new
  where
    old = lookupPrefix uris oldPr
    new = lookupPrefix uris newPr
```

To become effective, the aliases must be used in two different places. When a *LiteralQName* is instantiated and when an element is created by a literal result element. In the first case we have:

```
applyComputedQName :: ComputedQName -> Context -> QName
applyComputedQName (LiteralQName qName) ctx =
    lookupAlias (getAliases $ ctxGetStylesheet ctx) qName
-- case CompQName omitted
```

Where *lookupAlias* will either return the unchanged *QName* or a *QName* with a new namespace URI:

```
lookupAlias :: NSAliasing -> QName -> QName
lookupAlias nsm qn = mkQName (namePrefix qn) (localPart qn)
                       $ maybe (namespaceUri qn) id $ Map.lookup (namespaceUri qn) nsm
```

The full *createElement* function is shown below. The changes are marked bold. For non-literal result elements both the namespace URI and the namespace aliasing maps are empty.

```
createElement :: QName -> UriMapping -> NSAliasing -> [XmlTree] -> XmlTree
createElement name uris aliases fullcontent =
    mkElement name (nsAttrs ++ distinctAttribs) content
  where
    nsAttrs            = uriMap2Attrs $ aliasUriMapping aliases uris
    distinctAttribs    = nubBy eqAttr $ reverse attribs
    (attribs, content) = span (isAttr) fullcontent
    eqAttr node1 node2 = equivQName (fromJust $ getAttrName node1) (fromJust $ getAttrName node2)
```

The aliasing for the namespace mappings is implemented below.

```
aliasUriMapping :: NSAliasing -> UriMapping -> UriMapping
aliasUriMapping nsm = Map.map (\uri -> Map.findWithDefault uri uri nsm)
```

Namespace fixup

After processing the created XML tree has the following namespace related properties:

- All elements and attributes created from literal result elements have a proper list of namespace attributes. These can be highly redundant, since all in-scope namespace bindings are present on each element.
- All elements and attributes created by *xsl:element*, *xsl:attribute* and *xsl:copy* have qualified names with their correct namespace URIs, but there are no namespace attributes other than those which have been introduced by literal result elements.
- Therefore, the tree might contain elements and attributes with conflicting or undeclared namespace URIs. For example it is possible that two attributes of the same element have the same namespace prefixes, but different namespace URIs.

Namespace fixup has to be applied to create namespace attributes wherever needed, resolve name conflicts and clean up the redundant namespace declarations. The last step is optional but highly desirable, since the fully expanded namespace declaration make the resulting tree almost unreadable when outputted to a file.

Technically we have:

```
fixupNS :: XmlTree -> XmlTree
fixupNS = compressNS . disambigNS
```

The disambiguation must take place before the compression of redundant namespace declarations.

```
disambigNS :: XmlTree -> XmlTree
disambigNS =
    mapTreeCtx step $ Map.fromAscList [("xml", xmlNamespace), ("xmlns", xmlnsNamespace)]
  where
    step uris node
      | isElem node = let uris'               = uris `Map.union` getUriMap node
                          (newUris, newNode') = disambigElem uris' node in
                        (newUris, setUriMap newUris newNode')
      | otherwise   = (uris, node)
```

During disambiguation of an entire XML tree only the element nodes and their attribute lists are taken into account. The previously effective namespace declarations are passed downwards with the tree hierarchy. For each element disambiguation is performed by the function *disambigElem*. The result is a new set of effective namespace declarations and an element node for which all qualified names have consistent namespace URI and prefix parts. The new declarations are expanded to namespace attributes by *setUriMap*.

```
disambigElem :: UriMapping -> XNode -> (UriMapping, XNode)
disambigElem nsMap elem =
    (newNsMap, mkEmptyElement
                 (remapNsName newNsMap $ fromJust $ getElemName elem)
                 $ map (changeName $ remapNsName newNsMap) $ fromJust $ getAttrl elem )
  where
    newNsMap   = nsMap `Map.union` Map.fromAscList newTuples
    newTuples  = zip newPrefs $ nub newUris
    newUris    = filter (`notElem` oldUris) $ filter (not . null)
                   $ map namespaceUri $ mapMaybe getName
```

```
                (elem : map getNode (fromJust $ getAttrl elem))
    newPrefs   = filter (`notElem` oldPrefs) ["ns" ++ show i | i <- [1..]]
    oldPrefs   = Map.keys nsMap
    oldUris    = Map.elems nsMap
```

(D)isambigElem creates an infinite resource of fresh namespace prefixes *ns1*, *ns2*, *ns3*, ... from which all namespace prefixes which are already bound are removed. Then all unique namespace URIs which are not bound to any prefix in the current context are extracted from the qualified names of the attributes and the element. These are zipped together with the freshly created namespace prefixes. If we construct the union of the new namespace mappings and the current namespace mappings, we have a mapping for any namespace which is used. However, the qualified names can still be inconsistent and have to be corrected. Therefore the qualified names of the element and all of its attributes are *remapped*.

```
remapNsName :: UriMapping -> QName -> QName
remapNsName nsMap name =

    if maybe (nsUri=="") (== nsUri) luUri
    then name

    else mkQName newPref (localPart name) nsUri

  where
    luUri   = Map.lookup (namePrefix name) nsMap
    newPref = head $ (++ (error $ "int. error: No prefix for " ++ nsUri))
                $ Map.keys $ Map.filter (==namespaceUri name) nsMap
    nsUri   = namespaceUri name
```

If the namespace prefix and namespace URI are already consistent with the current namespace environment, or the name is unqualified, it can be kept unchanged. Otherwise any namespace prefix which maps to the names URI can be chosen as the new prefix. This is fully legal and in sync with the XSLT specification:

7.1.2 Creating elements with xsl:element [...] XSLT processors may make use of the prefix of the QName specified by the name attribute when selecting the prefix used for outputting the created element as XML; however, they are not required to do so.[...]

Please note that while we were allowed to choose an arbitrary prefix for these names, we are not allowed to change any of the bindings which were introduced by literal result elements. We are also not allowed to move the bindings arbitrarily upwards in the XML tree either. Our algorithm implicitly guarantees these requirements. The removal of superfluous namespace declarations in the aftermath is implemented below:

```
compressNS :: XmlTree -> XmlTree
compressNS = id
    mapTreeCtx compressElem $ Map.fromAscList [("xml", xmlNamespace), ("xmlns", xmlnsNamespace)]

compressElem :: UriMapping -> XNode -> (UriMapping, XNode)
compressElem uris node
  | isElem node = (newUris, changeAttrl (filter $ isImportant) node)
  | otherwise   = (uris, node)
  where
    newUris       = uris `Map.union` getUriMap node
    isImportant n = not (isNsAttr n)
                    || not ((localPart $ fromJust $ getAttrName n) `Map.member` uris)
```

This namespace fixup might become a part of the HXT core, if it proves to be useful for other applications. However it can only be added if it is useful exactly as it is specified here. The namespace handling requirement of XSLT are somewhat delicate and do not allow too much freedom, when it comes to the intrinsics of the disambiguation.

Conclusion

The result of this work is a working processor for a meaningful subset of XSLT implemented in Haskell. We hope that this processor will be actually used in the future and we will try to actively maintain and enhance it. Feedback from the Haskell community is highly appreciated.

The processor is written in a purely in Haskell 98 and does not make use of any advanced libraries. All XML operations are implemented in terms of the basic interfaces of the HXT library. The advanced DSL (domain specific language) interface of the library was not used. It might be possible to make the implementation even shorter and more concise by using this features; however, the basic implementation should enable interested volunteers to gain familiarity with the code rather quickly. The entire implementation has about 1800 lines of code and is thereby 192 times smaller than the XALAN Java implementation which has about 347000 lines of code. Of course, we could not implement the full functionality of such a huge tool within one master thesis; however, it should be possible to perform the most common kinds of stylesheet transformations within the subset of XSLT we implemented.

Some of the rules in the XSLT specification seem unnecessarily complicated. Neither XSLT nor XPath follow simple mathematical rules, which makes a formalization of the languages harder. There has been an approach to formalize a subset of XPath in *A formal semantic of patterns in XSLT* by Philip Wadler, but there is no complete formal description of the entire XPath or XSLT languages yet. Such a formalization would have been tremendously useful. In particular in Haskell the step from a formal definition to a working implementation of a language is often a small one.

The future direction of the XSLT processor should follow the demands of the Haskell community.

Bibliography

Apfel, Christine, *Konzeption und Design eines XSLT Prozessors unter dem Aspekt der funktionalen Programmierung mit Haskell*, Master thesis, Fachhochschule Wedel, 2002

Bird, Richard, *Introduction to Functional Programming* (second edition), Prentice Hall, 1998

English, Joe, *HXML* [http://www.flightlab.com/~joe/hxml/][9]

Gibbons, Jeremy, *Origami Programming*, in *the fun of programming*, Palgrave, 2003

Goerzen, John, *MissingH - Utilities for Haskell programmers* [http://directory.fsf.org/MissingH.html]

Jones, Mark, *Typing Haskell in Haskell*, Oregon Graduate Institute of Science and Technology, 1999

Kuseler Torben, *Konzeption und Implementation eines XPath-Moduls für die Haskell XML Toolbox*, Diplomarbeit, Fachhochschule Wedel, 2003

Lyons, Bob, *Universal Turing Machine in XSLT* ,[http://www.unidex.com/turing/utm.htm], 2001

Mangano, Sal, *XSLT Cookbook*, O'Reilly 2002

Pierce, Benjamin, *Types and Programming Languages*, MIT Press, 2002

Schmidt, Martin, *Design and Implementation of a validating XML parser in Haskell*, Master thesis, Fachhochschule Wedel, 2002

Thompson, Simon, *The craft of Functional Programming* (second edition), Addison Wesley, 1999

van Velzen, Danny, *An XSLT implementation in Haskell, Master thesis*, University of Amsterdam, 2001

Wadler, Philip, *A formal semantics of patterns in XSLT*, Bell Labs, 2000

Wallace, Malcolm, *HaXML* [http://www.cs.york.ac.uk/fp/HaXml/]

Wirth, Niklaus, *Grundlagen und Techniken des Compilerbaus*, Addison Wesley, 1996

XML Path Language, Version 1.0, W3C Recommendation, 1999

XML Path Language (XPath) 2.0, W3C Candidate Recommendation 8 June 2006

9 All internet sources last accessed on August 31, 2006

XSL Transformations (XSLT), Version 1.0, W3C Recommendation, 1999

XSL Transformations (XSLT), Version 2.0, W3C Candidate Recommendation 8 June 2006

Namespaces in XML 1.0 (Second Edition), W3C Recommendation 16 August 2006

Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation 04 February 2004

Appendix I - Unimplemented features

Forwards-compatible processing (2.5):

The current XSLT processor uses the xsl:version attribute to recognize an XML tree as an XSLT transformation. The value of it is ignored. To implement FCP this attribute must be expanded to entire XML tree of a single stylesheet during the *document level preprocessing* (p.24 ff) phase. The current XPath implementation, behaves according to FCP in that it only issues an error if an unsupported function is actually *evaluated*. Compiling an unsupported function must not and does not lead to an error.

Embedded stylesheets (2.7):

Currently not allowed. Probably not an important feature.

Numbering (7.7):

Not implemented. But: this feature can likely be implemented without causing any complicated interactions with other features; however, complicated internationalization issues might be involved depending the level of conformance which should be achieved.

Additional functions (12):

document (12.1)

Can be implemented in terms of *unsafePerformIO*. We can consider the external document as some sort of constant which is retrieved by some means. This is likely better than pervasively introducing the IO monad to the entire stylesheet application procedure.

Keys (12.2)

Will likely require an additional preprocessing step to collect all declared keys and evaluate them. *Is there a huge demand for this feature?*

format-number (12.3)

Partially implemented by XPath; however, none of the *xsl:decimal-format* parameters are passed to the XPath implementation yet.

current (12.4)

Will likely require to enhance the XPath context by some means.

unparsed-entity-uri

OK, now you got me. I've no idea what this is supposed to do...

generate-id

Can be implemented by the same mechanism which is used by XPath to compare nodes. See *getRelPosL* in the XPath function module.

system-property

Requires to extent the XPath context in such a way that the namespace environment can be accessed within the application of an XPath expression. So far, it is only passed to the XPath parser and is used to properly instantiate name tests.

Extensions & Fallback (14, 15)

The URIs of extension functions are properly expanded during document level preprocessing. This should make it fairly easy to implement fallback. The added XPath functions *function-available* and *element-available* would require access to the namespace environment just like the *system-property* function.

Output (16)

The implementation of *xsl:output* is optional. It is sufficient if an implementation is able to perform transformations on the level of XML trees. Our implementation does just that. The only concession for XML outputting is the namespace fixup we implemented. Within this section there is a *double optional* feature called *disable-output-escaping*, which can be added to any *xsl:text* instructions. This would be impossible to implement in terms of the current XML trees, as it would require us to mark any single text node in the result tree. The best thing is to ignore this *feature*.

Appendix II - Known limitations

Error handling

We bail out on the first error which occurs during the compilation or application of a stylesheet. This behavior is consistent with the XSLT specification and identical to XALAN's behavior. However, most HXT modules implement the collecting and accumulating of error messages. Collecting errors is certainly harder than bailing out. If it is a major advantage or even a disadvantage is mostly a matter of taste. We do issue error messages, if a required attribute is missing or has an illegal value, but superfluous attributes are simply ignored. This might lead to some surprises if an attribute name is misspelled. Therefore, it is usually advisable to use a second XSLT implementation to double-check the stylesheets. It would also be possible to perform some sort of Schema validation on the stylesheet prior to compilation.

Default namespaces

Default namespaces within XPath expressions and attribute value templates are currently not handled. Some subtleties are involved as we must distinguish between names which are supposed to represent attributes (default namespace not used) and name which are supposed to represent element (default namespace is used).

Efficiency of the matching procedure

The matching procedure as it is currently implemented is unreasonably inefficient. Some optimizations are sketched on page 36.

Binding variables to result tree fragments

We can only bind variables to the results of XPath expressions, that is to XPath values. The XSLT specification allows to bind variables to result tree fragments, that is to the result of an XSLT instruction. XSLT 1.0 specifies a new XPath data type for result tree fragments, while XSLT 2.0 treats result tree fragments as a node-set which consists of a freshly created root-node with the result tree fragments as its content. The XSLT 2.0 approach seems more regular and is backwards compatible. We would; however, need to find a way to assign a unique identifier to such an implicitly created root node. Otherwise conflicts could occur because XPath would not be able to distinguish between nodes which appear in the same positions in different tree fragments.

Appendix III - Alternative implementations

Parsing attribute value templates with regular expressions

```
import Text.Regex

parseAVT :: String -> StringExpr
parseAVT = StringExpr . concatExpr . splitAVT

splitAVT :: String -> [Expr]
splitAVT "" = []
splitAVT s  =

    if isJust parseLiteral then
       let (_,literal,rest,_) = fromJust parseLiteral
        in mkLiteralExpr (comprLit literal) : splitAVT rest

    else if isJust parseExprRe then
       let (_,_,rest,[expr]) = fromJust parseExprRe
        in parseExpr expr : splitAVT rest

    else
      error $ "illegal "++ s ++" substring found in attribute value template"

  where
    parseLiteral = matchRegexAll reLit  s
    parseExprRe  = matchRegexAll reExpr s
    reLit   = mkRegex "^([^{}]|{{|}})+"
    reExpr  = mkRegex "^{([^{}]+)}"
    comprLit ""           = ""
    comprLit ('{':'{':xs) = '{' : comprLit xs
    comprLit ('}':'}':xs) = '}' : comprLit xs
    comprLit (x:xs)       = x   : comprLit xs
```

(C)omprLit could be implemented in terms of subRegex, but that would be an overkill here. Again a simple replace function for strings (, or lists) would have helped.

Parsing attribute value templates with Parsec

The Grammar has been transformed to LL(1) by joining the cases for "{{" and "{expr}". Each quoted curly bracket introduces a new "literal expression". That means there are more expression fragments here than in the previous two versions.

```
import Text.ParserCombinators.Parsec.Prim        (Parser, parse, try, many, (<|>))
import Text.ParserCombinators.Parsec.Combinator  (many1)
import Text.ParserCombinators.Parsec.Char        (string, char, noneOf)

noBracketString :: Parser String
noBracketString = many1 (noneOf "{}")

parserAVT :: Parser Expr
parserAVT =    do -- '{' implies expression or quoted curly bracket "{{"
                 char '{'
                 (char '{' >> return (mkLiteralExpr "{"))
                  <|> do
                        exprStr <- noBracketString
                        char '}'
                        return $ parseExpr exprStr

          <|>  do -- quoted curly bracket "}}"
                 string "}}"
                 return $ mkLiteralExpr "}"

          <|>  do -- Literal
                 lit <- noBracketString
                 return $ mkLiteralExpr lit
```

```
parserAVTList :: Parser [Expr]
parserAVTList = many parserAVT

parseAVT :: String -> StringExpr
parseAVT avtStr = StringExpr $ concatExpr $ either (error . show) id parseResult
  where parseResult = parse parserAVTList ("attribute value template:"++avtStr) avtStr
```

Whitespace stripping on stylesheets in XSLT

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="*">
    <xsl:param name="strip" select="true()"/>
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates>
        <xsl:with-param name="strip"
           select="    ( local-name()!='text' or
                         namespace-uri()!='http://www.w3.org/1999/XSL/Transform' )
                   and not(@xml:space='preserve')
                   and ($strip or @xml:space='default') "/>
      </xsl:apply-templates>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="text()">
    <xsl:param name="strip" select="true()"/>
    <xsl:if test="not($strip) or string-length(normalize-space(.)) &gt; 0">
      <xsl:value-of select="." />
    </xsl:if>
  </xsl:template>

</xsl:transform>
```

The basic idea here is to pass a boolean *strip* parameter down which is by default *true* and changes its value only in case of an *xml:space* attribute or for the *xsl:text* instruction. The checking for the xsl:text name becomes unwieldy as we must take into account that stylesheets might use a prefix other than xsl to designate the XSLT namespace. All nodes are copied. Text nodes are only copied if they are non-empty or the current value of the *strip* argument is *false*. It is noticeable that this stylesheet does no longer work if we replace the (sub-) expression *not(@xml:space='preserve')* with *(@xml:space!='preserve').* That is a result of the very particular semantics of comparisons in XPath. Comparisons in XPath do not describe an equivalence relation. They are symmetric, but neither reflexive nor transitive!

Namespace expansion in XSLT'

If we *"unlock"* the namespace attributes in XPath[10], it is possible to perform some namespace related algorithms in XSLT. The example below demonstrates how the namespace attributes could be propagated to all its children. We could allow such transformations internally, if we wanted to implement parts of the XSLT processor in XSLT.

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="*">
    <xsl:param name="nsattrs" select="@xmlns:*"/>
    <xsl:copy>
      <xsl:copy-of select="@*|$nsattrs|@xmlns:*"/>
      <xsl:apply-templates>
        <xsl:with-param name="nsattrs" select="$nsattrs|@xmlns:*" />
      </xsl:apply-templates>
    </xsl:copy>
  </xsl:template>
</xsl:transform>
```

10 xmlns:* will always return the empty node set in XPath.

Appendix IV - Complete source code

Interface to the outside world

```
module XSLT
  ( module CompiledStylesheet
  , module Compilation
  , module Application )
  where

import CompiledStylesheet (CompiledStylesheet)

import Compilation (compileStylesheet,         -- :: XmlTree -> IO CompiledStylesheet
                    compileStylesheetFromUri ) -- :: String -> IO CompiledStylesheet

import Application (applyStylesheet,           -- CompiledStylesheet -> XmlTree -> [XmlTree]
        applyStylesheetWParams)    -- Map ExName Expr -> CompiledStylesheet -> XmlTree -> [XmlTree]
```

Central data structures

```
module CompiledStylesheet where
import Common
import Names
import Data.Maybe
import Data.List
import Data.Map(Map)
import qualified Data.Map as Map
import Maybe
import Control.Monad

-- -------------------
-- compiled-Stylesheet:

data CompiledStylesheet =
  CompStylesheet
    [MatchRule]
    (Map ExName NamedRule)
    (Map ExName Variable)
    (Map ExName [AttributeSet])
    [Strips]
    NSAliasing
  deriving Show

getMatchRules :: CompiledStylesheet -> [MatchRule]
getMatchRules (CompStylesheet matchRules _ _ _ _ _) = matchRules

getNamedRules :: CompiledStylesheet -> (Map ExName NamedRule)
getNamedRules (CompStylesheet _ namedRules _ _ _ _) = namedRules

getVariables :: CompiledStylesheet -> (Map ExName Variable)
getVariables (CompStylesheet _ _ variables _ _ _) = variables

getAttributeSets :: CompiledStylesheet -> Map ExName [AttributeSet]
getAttributeSets (CompStylesheet _ _ _ attrSets _ _) = attrSets

getStrips :: CompiledStylesheet -> [Strips]
getStrips (CompStylesheet _ _ _ _ strips _) = strips

getAliases :: CompiledStylesheet -> NSAliasing
getAliases (CompStylesheet _ _ _ _ _ aliases) = aliases

-- -------------------
-- common properties of match- and named- rules:

class Rule a where
  getRuleContent :: a -> Template
  getRuleParams  :: a -> [Variable]

-- -------------------
-- Match-Rules:

data MatchRule =
  MatRule MatchExpr
          Float           -- priority
          (Maybe ExName)  -- mode
          [MatchRule]     -- Imported rules only for xsl:apply-imports
          [Variable]      -- xsl:param list
          Template        -- content
  --deriving Show         -- output of imported Rules makes it unreadable

instance Show MatchRule where
  show (MatRule expr prio mode imprules params content) =
    "MkRule expr: "++ show expr ++"\n  prio: "++ show prio ++"\n  mode: "++ show mode
      ++"\n  no. imported rules: "++ show (length imprules) ++"\n  xsl-params: "++ show params
        ++"\n  content: "++ show content ++"\n"

instance Rule MatchRule where
  getRuleContent (MatRule _ _ _ _ _ c) = c
  getRuleParams  (MatRule _ _ _ _ p _) = p

getRulePrio :: MatchRule -> Float
getRulePrio (MatRule _ prio _ _ _ _) = prio

getRuleMode :: MatchRule -> Maybe ExName
getRuleMode (MatRule _ _ mode _ _ _) = mode

getRuleImports :: MatchRule -> [MatchRule]
getRuleImports (MatRule _ _ _ imports _ _) = imports

-- -------------------
-- Named-Rules:

data NamedRule = NamRule ExName [Variable] Template
  deriving Show

instance Rule NamedRule where
  getRuleContent (NamRule _ _ c) = c
  getRuleParams  (NamRule _ p _) = p

getRuleName :: NamedRule -> ExName
getRuleName (NamRule name _ _)  = name

-- -------------------
-- Variables

data Variable = MkVar
                  Bool        -- modus: False => xsl:variable, True => xsl:param
                  ExName      -- name
                  Expr        -- select
  deriving Show

getVarName :: Variable -> ExName
getVarName (MkVar _ name _) = name

getVarExpr :: Variable -> Expr
getVarExpr (MkVar _ _ expr) = expr

isParam :: Variable -> Bool
isParam (MkVar isP _ _) = isP

-- -------------------
-- Attribute sets:

newtype UsedAttribSets = UsedAttribSets [ExName]
  deriving Show

data AttributeSet = AttribSet ExName UsedAttribSets Template
  deriving Show

-- -------------------
-- Whitespace-stripping

type NTest = ExName

parseNTest :: UriMapping -> String -> NTest
parseNTest = parseExName

type Strips = Map NTest Bool
```

```haskell
lookupStrip :: ExName -> [Strips] -> Bool
lookupStrip name = head . (++[False]) . mapMaybe (lookupStrip1 name)

lookupStrip1 :: ExName -> Strips -> Maybe Bool
lookupStrip1 name spec =
    if      isJust nameMatch then nameMatch
    else if isJust prefMatch then prefMatch
    else if isJust globMatch then globMatch
    else Nothing
  where
    nameMatch = Map.lookup (       name             ) spec
    prefMatch = Map.lookup (ExName "*"  $ exUri name) spec
    globMatch = Map.lookup (ExName "*"  ""          ) spec

feedSpaces :: Bool -> [NTest] -> Strips -> Strips
feedSpaces strip tests =
    Map.unionWithKey feedErr $ Map.fromListWithKey feedErr $ zip tests $ repeat strip
  where
    feedErr k = error $ "Ambiguous strip- or preserve-space rules for " ++ show k

feedStrips, feedPreserves :: [NTest] -> Strips -> Strips
feedStrips    = feedSpaces True
feedPreserves = feedSpaces False

stripDocument :: [Strips] -> XmlTree -> XmlTree
stripDocument strips =
  stripSpaces (\_ n -> lookupStrip (mkExName $ fromJust $ getElemName n) strips) False

stripStylesheet :: XmlTree -> XmlTree
stripStylesheet = stripSpaces isStrip True
  where
    isStrip strip' node =
         not (isElemType xsltText node)
      && (maybe strip' (=="default") $ tryFetchAttribute node xmlSpace)

stripSpaces :: (Bool -> XNode -> Bool) -> Bool -> XmlTree -> XmlTree
stripSpaces f def =
    fromJustErr "stripSpaces (internal error)" . filterTreeCtx step def
  where
    step strip node
     | isElem node           = (f strip node, True)
     | isWhitespaceNode node = (strip       , not strip)
     | otherwise             = (strip       , True)

-- -------------------
-- Namespace aliases and exclusion

-- Map a namespace URI to an alias URI
type NSAliasing = Map String String

addAlias :: UriMapping -> String -> String -> NSAliasing -> NSAliasing
addAlias uris oldPr newPr =
    Map.insertWith (error $ "duplicate mapping for " ++ old) old new
  where
    old = lookupPrefix uris oldPr
    new = lookupPrefix uris newPr

-- lookup an alias in a namespace-mapping.
-- returns the original name, if there is no alias for that name.
lookupAlias :: NSAliasing -> QName -> QName
lookupAlias nsm qn = mkQName (namePrefix qn) (localPart qn)
                       $ maybe (namespaceUri qn) id $ Map.lookup (namespaceUri qn) nsm

aliasUriMapping :: NSAliasing -> UriMapping -> UriMapping
aliasUriMapping nsm = Map.map (\uri -> Map.findWithDefault uri uri nsm)

-- -------------------
-- Templates:

data Template =
    TemplComposite [Template]
  | TemplForEach SelectExpr [SortKey] Template
  | TemplChoose [When]
  | TemplMessage Bool     -- halt?
                 Template
  | TemplElement ComputedQName
                 UriMapping
                 UsedAttribSets
                 Template
  | TemplAttribute ComputedQName
                   Template
  | TemplText      String
  | TemplValueOf StringExpr
  | TemplComment Template
  | TemplProcInstr StringExpr
                   Template
  | TemplApply (Maybe SelectExpr)
               (Maybe ExName)  -- mode
               (Map ExName Variable)
               [SortKey]
  | TemplApplyImports
  | TemplVariable Variable
  | TemplCall ExName
              (Map ExName Variable)
  | TemplCopy UsedAttribSets
              Template
  | TemplCopyOf Expr
  deriving Show

data SortKey = SortK StringExpr -- select
                     StringExpr -- data-type: number or text(default)
                     StringExpr -- order: ascending(default) or descending
  deriving Show

data When = WhenPart TestExpr Template
  deriving Show

data ComputedQName = LiteralQName QName
                   | CompQName UriMapping -- namespace-env
                               StringExpr -- name
                               StringExpr -- namespace
  deriving Show

-- -------------------
-- different kinds of expressions
newtype SelectExpr = SelectExpr Expr   deriving Show
newtype TestExpr   = TestExpr   Expr   deriving Show
newtype StringExpr = StringExpr Expr   deriving Show
newtype MatchExpr  = MatchExpr  Expr   deriving Show
```

Stylesheet compilation

```
module Compilation
  ( compileStylesheet,         -- :: XmlTree -> IO CompiledStylesheet
    compileStylesheetFromUri   -- :: String -> IO CompiledStylesheet
   )
  where
import Common
import Names
import CompiledStylesheet
import Control.Monad
import Control.Exception
import Data.Maybe
import Data.Either
import Data.List
import qualified Data.Map as Map
import Data.Map(Map)
import qualified Data.Set as Set
import Text.ParserCombinators.Parsec.Prim(runParser)
import Control.Arrow
import Debug.Trace

-- No deep meaning just a shortcut notation for a *very* common expression...
infixl 9 ><
(><) :: XmlNode n => (UriMapping -> a ) -> n -> a
f><node = f $ getUriMap node

-- --------------------------

parseExpr :: UriMapping -> String -> Expr
parseExpr uris selectStr = either (error.show) id parseResult
  where parseResult = runParser parseXPath uris ("select-expr:"++selectStr) selectStr
```

```haskell
parseSelect :: UriMapping -> String -> SelectExpr
parseSelect uris = SelectExpr . parseExpr uris

parseTest :: UriMapping -> String -> TestExpr
parseTest uris = TestExpr . mkBoolExpr . parseExpr uris

parseStringExpr :: UriMapping -> String -> StringExpr
parseStringExpr uris = StringExpr . mkStringExpr . parseExpr uris

parseMatch :: UriMapping -> String -> MatchExpr
parseMatch uris str =
    if isMatchExpr expr
    then MatchExpr expr
    else error $ str ++ " is not a legal match-expression"
  where
    expr = parseExpr uris str

-- -------------------------

parseAVT :: UriMapping -> String -> StringExpr
parseAVT uris str =
    StringExpr $ concatExpr $ splitAVT str ""
  where

    splitAVT :: String -> String -> [Expr]
    splitAVT ""           acc = acc2lit acc
    splitAVT ('{':'{':xs) acc = splitAVT xs $ '{':acc
    splitAVT ('}':'}':xs) acc = splitAVT xs $ '}':acc
    splitAVT ('{':xs)     acc = let (body, rest) = span (`notElem` "{}") xs in
                                  if not (null rest) && head rest == '}'
                                  then acc2lit acc ++ parseExpr uris body : splitAVT (tail rest) ""
                                  else error $ "Unterminated expression " ++ xs ++ " in AVT."
    splitAVT ('}':_)      _   = error $ "deserted '}' in AVT."
    splitAVT (x:xs)       acc = splitAVT xs $ x:acc

    acc2lit :: String -> [Expr]
    acc2lit ""  = []
    acc2lit acc = [mkLiteralExpr $ reverse acc]

-- -------------------------

-- extract ComputedQName from "name" and "namespace" AVTs of an xsl:element- or xsl-attribute-node
compileComputedQName :: XmlTree -> ComputedQName
compileComputedQName node =
    (CompQName><node) nameAVT nsAVT
  where
    nameAVT  = parseAVT><node $ fetchAttribute node xsltName
    nsAVT    = parseAVT><node $ fetchAttributeWDefault node xsltNamespace ""

-- -------------------------

compileComposite :: [XmlTree] -> Template
compileComposite = TemplComposite . map (compileTemplate . return)

compileMessage :: XmlTree -> Template
compileMessage node = TemplMessage halt content
  where halt     = termAttr == "yes"
        termAttr = fetchAttributeWDefault node xsltTerminate "no"
        content  = compileTemplate (getChildren node)

compileForEach :: XmlTree -> Template
compileForEach node = TemplForEach expr sorting template
  where expr       = parseSelect><node $ fetchAttribute node xsltSelect
        sorting    = map compileSortKey srt
        template   = compileTemplate cnt
        (srt, cnt) = partition (isElemType xsltSort) $ getChildren node

compileChoose :: XmlTree -> Template
compileChoose node = TemplChoose whenParts
  where whenParts  = map compl children
        children   = filter isElem (getChildren node)
        compl node' = let elemName = fromJust $ getElemName node' in
                        if      equivQName elemName xsltWhen      then compileWhen node'
                        else if equivQName elemName xsltOtherwise then compileOtherwise node'
                        else error $ "No elements of type " ++ show elemName
                                     ++ " allowed within xsl-choose template!"
```

```
compileWhen :: XmlTree -> When
compileWhen node = WhenPart expr $ compileTemplate $ getChildren node
  where expr     = parseTest><node $ fetchAttribute node xsltTest

-- Otherwise is treated as a when-Part with node-test "true()"
compileOtherwise :: XmlTree -> When
compileOtherwise node = WhenPart (TestExpr mkTrueExpr) $ compileTemplate $ getChildren node

-- "if" is treated as a convenience-form of choose with exactly one "when"-Part
compileIf :: XmlTree -> Template
compileIf = TemplChoose . return . compileWhen

-- ---------------------------------

parseExNames :: UriMapping -> String -> [ExName]
parseExNames urm = map (parseExName urm) . words

compileElement :: XmlTree -> Template
compileElement node =
    TemplElement compQName Map.empty attribSets template
  where
    compQName   = compileComputedQName node
    attribSets  = UsedAttribSets $ parseExNames><node
                  $ fetchAttributeWDefault node xsltUseAttributeSets ""
    template    = compileTemplate (getChildren node)

compileAttribute :: XmlTree -> Template
compileAttribute node =
    TemplAttribute (compileComputedQName node) $ compileTemplate (getChildren node)

-- compiles xsl:text
compileText :: XmlTree -> Template
compileText = TemplText . collectTextnodes . getChildren

-- compiles textNode
compileTextnode :: XmlTree -> Template
compileTextnode = TemplText . fromJust . getText

compileValueOf :: XmlTree -> Template
compileValueOf node =
    TemplValueOf $ parseStringExpr><node $ fetchAttribute node xsltSelect

compileComment :: XmlTree -> Template
compileComment = TemplComment . compileTemplate . getChildren

compileProcInstr :: XmlTree -> Template
compileProcInstr node =
   TemplProcInstr name content
  where
    name    = parseAVT><node  $ fetchAttribute node xsltName
    content = compileTemplate $ getChildren node

-- ---------------------------------

compileLiteralResultElement :: XmlTree -> Template
compileLiteralResultElement node =
    TemplElement compQName nsUris attribSets content
  where
    nsUris             = extractAddUris node
    compQName          = LiteralQName   $ fromJust $ getElemName node
    attribSets         = UsedAttribSets $ parseExNames><node $ attrSetsStr
    attrSetsStr        = fetchAttributeWDefault node xsltUseAttributeSetsLRE ""
    content            = TemplComposite $ attributes ++ [template]
    attributes         = mapMaybe (compileLREAttribute><node) $ fromJust $ getAttrl node
    template           = compileTemplate (getChildren node)

compileLREAttribute :: UriMapping -> XmlTree -> Maybe Template
compileLREAttribute uris node =
    if isSpecial
      then Nothing
      else Just $ TemplAttribute (LiteralQName name) val
  where
    isSpecial = namespaceUri name `elem` [xsltUri, xmlnsNamespace]
    name      = fromJust $ getAttrName node
    val       = TemplValueOf $ parseAVT uris $ collectTextnodes $ getChildren node

-- ---------------------------------
```

```
compileApplyTempl :: XmlTree -> Template
compileApplyTempl node =
    TemplApply expr mode args sorting
  where
    expr    = liftM (parseSelect><node) $ tryFetchAttribute node xsltSelect
    mode    = liftM (parseExName><node) $ tryFetchAttribute node xsltMode
    args    = compileVariables          $ filter (isElemType xsltWithParam) $ par
    sorting =  map compileSortKey srt
    (srt,par) =  partition (isElemType xsltSort) $ getChildren node

compileApplyImports :: XmlTree -> Template
compileApplyImports node = TemplApplyImports

compileCallTempl :: XmlTree -> Template
compileCallTempl node =
    TemplCall name args
  where
    name = parseExName><node $ fetchAttribute node xsltName
    args = compileVariables  $ filter (isElemType xsltWithParam) $ getChildren node

compileTemplVariable :: XmlTree -> Template
compileTemplVariable = TemplVariable . compileVariable

-- -----------------------------------

compileCopy :: XmlTree -> Template
compileCopy node =
    TemplCopy attribSets $ compileTemplate (getChildren node)
  where
    attribSets  = UsedAttribSets $ parseExNames><node $ fetchAttributeWDefault node
xsltUseAttributeSets ""

compileCopyOf :: XmlTree -> Template
compileCopyOf node = TemplCopyOf $ parseExpr><node $ fetchAttribute node xsltSelect

-- -----------------------------------

compileTemplate :: [XmlTree] -> Template
compileTemplate [node]       =
   if isElem node
   then let elemName = fromJust $ getElemName node in
        if      equivQName elemName xsltMessage        then compileMessage       node
        else if equivQName elemName xsltForEach        then compileForEach       node
        else if equivQName elemName xsltChoose         then compileChoose        node
        else if equivQName elemName xsltIf             then compileIf            node
        else if equivQName elemName xsltElement        then compileElement       node
        else if equivQName elemName xsltAttribute      then compileAttribute     node
        else if equivQName elemName xsltText           then compileText          node
        else if equivQName elemName xsltValueOf        then compileValueOf       node
        else if equivQName elemName xsltComment        then compileComment       node
        else if equivQName elemName xsltProcInstr      then compileProcInstr     node
        else if equivQName elemName xsltApplyTemplates then compileApplyTempl    node
        else if equivQName elemName xsltApplyImports   then compileApplyImports  node
        else if equivQName elemName xsltCallTemplate   then compileCallTempl     node
        else if equivQName elemName xsltVariable       then compileTemplVariable node
        else if equivQName elemName xsltCopy           then compileCopy          node
        else if equivQName elemName xsltCopyOf         then compileCopyOf        node

        -- no other xslt elements allowed here:
        else if namespaceUri elemName == xsltUri
        then error $ "xslt-element " ++ localPart elemName ++ " not allowed within this context."

        -- for now all other elements will be considered as Literal Result Elements
        else compileLiteralResultElement node

   else if isText node then compileTextnode node

   else
       error $ "Unsupported node-type in xslt sheet: " ++ show (getNode node)
compileTemplate list = compileComposite list

-- -----------------------------------
-- Assembling of the entire stylesheet

assembleStylesheet :: XmlTree -> [CompiledStylesheet] -> CompiledStylesheet
assembleStylesheet xslNode imports=
    CompStylesheet matchRules namedRules variables attsets strips aliases
  where
    -- entire contents:
    (namedRules,
     matchRules)           = assembleRules ruleElems importedMatchRules importedNamedRules
    variables              = assembleVariables varElems importedVariables
    attsets                = assembleAttrSets attsetElems importedAttribSets
    strips                 = assembleStrips stripElems preserveElems importedStrips
    aliases                = assembleAliases nsAliasElems importedAliases

    -- element content:
    (nsAliasElems, r5)     = partition (isElemType xsltNamespaceAlias) r4
    (ruleElems, r4)        = partition (isElemType xsltTemplate) r3
    (varElems, r3)         = partition (\node ->    isElemType xsltVariable node
                                                 || isElemType xsltParam node) r2
    (attsetElems, r2)      = partition (isElemType xsltAttributeSet) r1
    (preserveElems, r1)    = partition (isElemType xsltPreserveSpace) r0
    (stripElems, r0)       = partition (isElemType xsltStripSpace) $ getChildren xslNode

    -- imported stuff:
    importedAttribSets     = map getAttributeSets imports
    importedVariables      = map getVariables revImports
    importedNamedRules     = map getNamedRules revImports
    importedMatchRules     = concatMap getMatchRules revImports
    importedStrips         = concatMap getStrips revImports
    importedAliases        = map getAliases revImports
    revImports             = reverse imports

assembleRules :: [XmlTree] -> [MatchRule] -> [Map ExName NamedRule]
                                                              -> (Map ExName NamedRule, [MatchRule])
assembleRules nodes importedMatches importedProcs =
    (resProcs, resMatches)
  where

    -- matches:
    resMatches        = localMatches ++ importedMatches
    localMatches      = reverse $ sortBy cmp matches
    cmp rulA rulB     = compare (getRulePrio rulA) (getRulePrio rulB)

    -- procedures:
    resProcs          = Map.unions (localProcs:importedProcs)
    localProcs        = foldl ins Map.empty procs
    ins map rule      = Map.insertWith (error $ "named-rule "++ show (getRuleName rule)
                                                ++" is already defined on this level")
                                        (getRuleName rule) rule map

    -- compile all xsl:template's:
    (procs, matches) = catMaybes *** concat $ unzip $ map (compileRule importedMatches) nodes

assembleVariables :: [XmlTree] -> [(Map ExName Variable)] -> (Map ExName Variable)
assembleVariables varElems = Map.unions . (compileVariables varElems:)

assembleAttrSets :: [XmlTree] -> [Map ExName [AttributeSet]] -> Map ExName [AttributeSet]
assembleAttrSets attsetElems =
    foldr (Map.unionWith (++)) localAttribSets
  where
    localAttribSets        = foldr insertAs Map.empty
                             $ map compileAttributeSet attsetElems
    insertAs as@(AttribSet name _ _) = Map.insertWith (++) name [as]

assembleStrips :: [XmlTree] -> [XmlTree]-> [Strips] -> [Strips]
assembleStrips stripElems preserveElems =
    (localStrips :)
  where
    localStrips = feedStrips (concatMap compileStrips stripElems)
                $ feedPreserves (concatMap compilePreserves preserveElems)
                $ Map.empty

assembleAliases :: [XmlTree] -> [NSAliasing] -> NSAliasing
assembleAliases nsAliasElems =
    Map.unions . (localAliases:)
  where
    localAliases           = foldr addAlias' Map.empty nsAliasElems
    addAlias' node         = uncurry (addAlias><node) $ compileAlias node

-- -----------------------------------
```

```
isStylesheetElem :: XmlTree -> Bool
isStylesheetElem node =
  (isElemType xsltTransform node || isElemType xsltStylesheet node) && hasAttribute node xsltVersion

isLREstylesheet :: XmlTree -> Bool
isLREstylesheet node  = hasAttribute node xsltVersionLRE

lre2template :: XmlTree -> XmlTree
lre2template = mkElement xsltTemplate [mkAttr xsltMatch [mkText "/"]] . return

lre2stylesheet :: XmlTree -> XmlTree
lre2stylesheet = mkElement xsltTransform [] . return . lre2template

-- ----------------------------------
-- Stylesheet compilation in the IO Monad:

compileStylesheetFromUri :: String -> IO CompiledStylesheet
compileStylesheetFromUri = compileStylesheetFromUriWIncStk []

compileStylesheetFromUriWIncStk :: [String] -> String -> IO CompiledStylesheet
compileStylesheetFromUriWIncStk incstack uri = readStylesheetWIncStk incstack uri >>=
compileStylesheetWIncStk (uri:incstack)

readStylesheetWIncStk :: [String] -> String -> IO XmlTree
readStylesheetWIncStk incstack uri =
  if uri `elem` incstack
  then error $ "Error: " ++ uri ++ " is recursively imported/included."
          ++ concatMap ("\n  imported/included from: " ++) incstack
  else readDocumentIO [(a_preserve_comment, "0")] uri >>= return . prepareXSLTDocument

compileStylesheet ::  XmlTree -> IO CompiledStylesheet
compileStylesheet = compileStylesheetWIncStk [] . prepareXSLTDocument

compileStylesheetWIncStk :: [String] -> XmlTree -> IO CompiledStylesheet
compileStylesheetWIncStk incstack node =

  -- ======= 1: simplified syntax
  if isLREstylesheet xslNode then
    return $ assembleStylesheet (lre2stylesheet xslNode) []

  -- ======= 2: regular syntax
  else if isStylesheetElem xslNode then
    do
      -- ======= 2.1: gather included stylesheets
      expandedContent <- expandIncludes incstack content

      -- ======= 2.2: compile imported stylesheets
      (imps, rest) <- return $ partition (isElemType xsltImport) expandedContent
      imports      <- mapM (compileStylesheetFromUriWIncStk incstack . getHRef) $ imps

      -- ======= 2.3: compile content
      expandedStylesheet <- return $ setChildren rest xslNode
      return $ assembleStylesheet expandedStylesheet imports

  -- ======= 3: unknown document type:
  else error "Expected: Either xsl:stylesheet/xsl:transform or simplified syntax"

  where
    content     = getChildren xslNode
    (xslNode:_) = filter isElem $ getChildren $ node
    getHRef     = flip fetchAttribute xsltHRef


expandIncludes :: [String] -> [XmlTree] -> IO [XmlTree]
expandIncludes incstack = liftM concat . mapM (expandInclude incstack) . filter isElem

expandInclude :: [String] -> XmlTree -> IO [XmlTree]
expandInclude incstack node =
  if isElemType xsltInclude node
    then
    do
       -- ======= read include-stylesheet and extract stylesheet node
       href        <- return $ fetchAttribute node xsltHRef
       docNode     <- readStylesheetWIncStk incstack href
       (xslNode:_) <- return $ filter isElem $ getChildren docNode

       -- ======= check for simplified syntax
       if isLREstylesheet xslNode
         then return [lre2template xslNode]

       -- ======= check for xsl:stylesheet or xsl:transform
         else if isStylesheetElem xslNode
                 then expandIncludes (href:incstack) $ getChildren xslNode

       -- ======= include file has unknown type
                 else error $ "Error: Included file " ++ href ++ " is not a stylesheet"
    else return [node]

-- ----------------------------------

compileRule :: [MatchRule] -> XmlTree -> (Maybe NamedRule, [MatchRule])
compileRule imports node =

    if isNothing match && isNothing name
    then error "Error: Bogus rule (xsl:template) with neither match nor name attribute is illegal"

    else if isJust mode && isNothing match
    then error "Error: Bogus mode attribute on none-match rule is illegal"

    else if isJust priority && isNothing match
    then error "Error: Bogus priority attribute on none-match rule is illegal"

    else
      (
        liftM (\name -> NamRule name params template) name
      , concat $ maybeToList $ liftM (assembleMatchRule priority mode imports params template) match
      )

  where
    match      = liftM (parseMatch><node)  $ tryFetchAttribute node xsltMatch
    name       = liftM (parseExName><node) $ tryFetchAttribute node xsltName
    priority   = liftM read                $ tryFetchAttribute node xsltPriority
    mode       = liftM (parseExName><node) $ tryFetchAttribute node xsltMode
    template   = compileTemplate content
    params     = map compileVariable paramsXml
    (paramsXml, content) =
                  partition (isElemType xsltParam) $ getChildren node

assembleMatchRule :: Maybe Float -> Maybe ExName -> [MatchRule] -> [Variable] -> Template ->
MatchExpr -> [MatchRule]
assembleMatchRule pri m imp par tmpl mtch@(MatchExpr expr) =
    if isJust pri
    then return $ MatRule mtch (fromJust pri) m imp par tmpl
    else map expand $ splitMatchByPrio expr
  where
    expand (pri, mtch) = MatRule (MatchExpr mtch) pri m imp par tmpl

-- ----------------------------------

compileVariables :: [XmlTree] -> Map ExName Variable
compileVariables nodes =
    foldl insertVar Map.empty $ varList
  where
    varList                = map compileVariable $ nodes
    insertVar map var      = Map.insertWith (error $ "parameter or variable "
                                    ++ show (getVarName var) ++" is already defined on this level")
                                           (getVarName var) var map

compileVariable :: XmlTree -> Variable
compileVariable node =
    MkVar modus name expr
  where
    modus   = isElemType xsltParam node
    name    = parseExName><node $ fetchAttribute node xsltName
    expr    = parseExpr><node   $ fetchAttributeWDefault node xsltSelect "''"

-- ----------------------------------

compileAttributeSet :: XmlTree -> AttributeSet
compileAttributeSet node =
    AttribSet name usedsets template
  where
    name     = parseExName><node $ fetchAttribute node xsltName
    usedsets = UsedAttribSets    $ parseExNames><node $ fetchAttributeWDefault node
```

```
                                                              xsltUseAttributeSets ""
    template = compileTemplate   $ filter (isElemType xsltAttribute) $ getChildren node

-- -----------------------------------

compileSortKey :: XmlTree -> SortKey
compileSortKey node =
    SortK expr dataType order
  where
    expr     = parseStringExpr><node $ fetchAttributeWDefault node xsltSelect "."
    dataType = parseAVT><node        $ fetchAttributeWDefault node xsltDataType "text"
    order    = parseAVT><node        $ fetchAttributeWDefault node xsltOrder "ascending"

-- -----------------------------------

parseNTests :: UriMapping -> String -> [NTest]
parseNTests uris = map (parseNTest uris) . words

compileStrips,compilePreserves :: XmlTree -> [NTest]
compileStrips node = parseNTests><node $ fetchAttribute node xsltElements
compilePreserves = compileStrips

-- -----------------------------------

compileAlias :: XmlTree -> (String, String)
compileAlias node =
  (fetchAttribute node xsltStylesheetPrefix, fetchAttribute node xsltResultPrefix)

-- -----------------------------------
-- Document level preprocessing

prepareXSLTDocument :: XmlTree -> XmlTree
prepareXSLTDocument = expandExEx . expandNSDecls . stripStylesheet . removePiCmt

removePiCmt :: XmlTree -> XmlTree
removePiCmt = fromJustErr "XSLT: No root element" . filterTree (\n -> not (isPi n) && not (isCmt n))

-- Expand exclude-result-prefixes AND extension-element-prefixes
expandExEx :: XmlTree -> XmlTree
expandExEx = mapTreeCtx expandExExElem ([xsltUri,xmlNamespace,xmlnsNamespace],[])

expandExExElem :: ([String], [String]) -> XNode -> (([String], [String]), XNode)
expandExExElem c@(excl, ext) node
  | isElem node = ((exclAcc, extAcc), nodeNew)
  | otherwise   = (c, node)
  where
    nodeNew    = setAttribute nameExcl (unwords exclAcc)
                 $ setAttribute nameExt (unwords extAcc) node
    exclAcc    = exclNew ++ excl
    extAcc     = extNew  ++ ext
    exclNew    = extNew  ++ (parsePreList><node $ fetchAttributeWDefault node nameExcl "")
    extNew     =             parsePreList><node $ fetchAttributeWDefault node nameExt  ""
    (nameExcl,
     nameExt)  = if (namespaceUri $ fromJust $ getElemName node) == xsltUri
                  then (xsltExlcudeResultPrefixes   , xsltExtensionElementPrefixes   )
                  else (xsltExlcudeResultPrefixesLRE, xsltExtensionElementPrefixesLRE)

-- parse a prefix list, create a list of uris:
-- "pre1 pre2 pre3" -> ["pre1.uri","pre2.uri","pre3.uri"]
parsePreList :: UriMapping -> String -> [String]
parsePreList uris = map (lookupPrefix uris) . words

-- -----------------------------------
-- Extraction of contextual Information from an XML-Node

extractAddUris :: XmlTree -> UriMapping
extractAddUris node =
    (Map.filter (`notElem` exclUris))><node
  where
    exclUris = words $ fetchAttributeWDefault node xsltExlcudeResultPrefixesLRE ""
```

Stylesheet application

```
module Application
  ( applyStylesheet,       -- CompiledStylesheet -> XmlTree -> [XmlTree]
    applyStylesheetWParams -- Map ExName Expr -> CompiledStylesheet -> XmlTree -> [XmlTree]
  )
  where

import Common
import Names
import CompiledStylesheet
import Control.Exception
import Data.Maybe
import Data.Either
import Data.List
import qualified Data.Map as Map
import Data.Map(Map)
import Debug.Trace(trace)
import Data.Char

type VariableSet = Map ExName XPathValue
type ParamSet = VariableSet

data Context = Ctx NavXmlTree                -- current node
                   [NavXmlTree]              -- current node list
                   Int                       -- pos. of curr-node 1..length
                   Int                       -- length of node list
                   VariableSet               -- glob. Var
                   VariableSet               -- loc. Var
                   CompiledStylesheet        -- The stylesheet which is being applied
                   (Maybe MatchRule)         -- Just last applied rule, Nothing within xsl:for-each
               | CtxEmpty -- The empty-context, indicates that transformation branch is finished

ctxGetNode :: Context -> NavXmlTree
ctxGetNode CtxEmpty = error "ctxGetNode: Internal error attempt to access the empty context"
ctxGetNode (Ctx node _ _ _ _ _ _ _) = node

ctxGetNodes :: Context -> [NavXmlTree]
ctxGetNodes CtxEmpty = []
ctxGetNodes (Ctx _ nodes _ _ _ _ _ _) = nodes

ctxGetStylesheet :: Context -> CompiledStylesheet
ctxGetStylesheet CtxEmpty = error "ctxGetStylesheet: Internal error ..."
ctxGetStylesheet (Ctx _ _ _ _ _ _ stylesheet _) = stylesheet

ctxGetRule :: Context -> Maybe MatchRule
ctxGetRule CtxEmpty = Nothing
ctxGetRule (Ctx _ _ _ _ _ _ _ rule) = rule

ctxSetNodes :: [NavXmlTree] -> Context -> Context
ctxSetNodes _ CtxEmpty = error "ctxSetNodes: Internal error..."
ctxSetNodes [] _       = CtxEmpty
ctxSetNodes nodes ctx@(Ctx _ _ _ _ globVars locVars cs rl) =
  Ctx (head nodes) nodes 1 (length nodes) globVars locVars cs rl

ctxSetRule :: Maybe MatchRule -> Context -> Context
ctxSetRule _ CtxEmpty = error "ctxSetRule: Internal error attempt to access the empty context"
ctxSetRule rule ctx@(Ctx node nodes pos len globVars locVars cs _) =
  Ctx node nodes pos len globVars locVars cs rule

addVariableBinding :: ExName -> XPathValue -> Context -> Context
addVariableBinding name val (Ctx node nodes pos len globVars locVars cs rl) =
    Ctx node nodes pos len globVars locVarsNew cs rl
  where locVarsNew = Map.insertWith (errF) name val locVars
        errF       = error $ "Local variable or parameter " ++ show name
                             ++ " is already bound in this context"

clearLocalVariables :: Context -> Context
clearLocalVariables CtxEmpty = CtxEmpty
clearLocalVariables (Ctx node nodes pos len globVars _ cs rl) =
  (Ctx node nodes pos len globVars Map.empty cs rl)

processContext :: Context -> (Context->[XmlTree]) -> [XmlTree]
processContext CtxEmpty f = []
processContext ctx@(Ctx node nList pos len gloVar locVar cs rl) f
 | pos > len = []
 | otherwise = f ctx ++ processContext (Ctx (nList!!pos) nodeList (pos+1) len gloVar locVar cs rl) f

-- ----------------

evalXPathExpr :: Expr -> Context -> XPathValue
evalXPathExpr expr ctx@(Ctx node _ pos len globVars locVars _ _) =
    filterXPath $ evalExpr (vars,[]) (pos, len, node) expr (XPVNode [node])
  where
    filterXPath (XPVError err)    = error err
    filterXPath (XPVNode nodes)   = XPVNode $ (\x -> fst x ++ snd x)
                                            $ partition (isAttr . subtreeNT) nodes
    filterXPath xpv               = xpv
    vars                          = map (\(name, val) -> ((exUri name, exLocal name), val)) varList
    varList                       = Map.toAscList $ locVars `Map.union` globVars

applySelect :: SelectExpr -> Context -> [NavXmlTree]
applySelect (SelectExpr expr) ctx =
    extractNodes xpathResult
  where
    xpathResult                  = evalXPathExpr expr ctx
    extractNodes (XPVNode nodes) = nodes
    extractNodes r               = error $ "XPATH-Expression in select or match attribute returned "
                                         ++"a value of the wrong type (" ++ take 15 (show r) ++ "...)"

applyTest :: TestExpr -> Context -> Bool
applyTest (TestExpr expr) ctx = bool
  where (XPVBool bool) = evalXPathExpr expr ctx

applyStringExpr :: StringExpr -> Context -> String
applyStringExpr (StringExpr expr) ctx = string
  where (XPVString string) = evalXPathExpr expr ctx

applyMatch :: MatchExpr -> Context -> Bool
applyMatch (MatchExpr expr) ctx =
    matchBySelect (SelectExpr expr) (ctxGetNode ctx) ctx
  where
    matchBySelect :: SelectExpr -> NavXmlTree -> Context -> Bool
    matchBySelect _ _ CtxEmpty = False
    matchBySelect expr matchNode ctx =
       if matchNode `isNotInNodeList` applySelect expr ctx
       then matchBySelect expr matchNode $ ctxSetNodes (maybeToList $ upNT $ ctxGetNode ctx) ctx
       else True

-- ------------------------------------

applyComputedQName :: ComputedQName -> Context -> QName

applyComputedQName (LiteralQName qName) ctx =
    lookupAlias (getAliases $ ctxGetStylesheet ctx) qName

applyComputedQName (CompQName uris nameATV nsATV) ctx =
    if null nsuri && not (null pref)
    then mkQName pref loc $ lookupPrefix uris pref
    else mkQName pref loc nsuri
  where
    nsuri         = applyStringExpr nsATV ctx
    (pref, loc)   = if null loc' then ("", pref')
                                 else (pref', tail loc')
    (pref', loc') = span (/=':') $ applyStringExpr nameATV ctx

-- ------------------------------------

applyComposite :: Template -> Context -> [XmlTree]
applyComposite (TemplComposite templates) ctx =
    concat $ reverse $ fst $ foldl applyElem ([], ctx) templates
  where
    applyElem :: ([[XmlTree]], Context) -> Template -> ([[XmlTree]], Context)
    applyElem (nodes, ctx) (TemplVariable v) = (nodes, processLocalVariable v Map.empty ctx)
    applyElem (nodes, ctx) t                 = (applyTemplate t ctx:nodes, ctx)


applyForEach :: Template -> Context -> [XmlTree]
applyForEach (TemplForEach expr sorting template) ctx =
    processContext sortedCtx $ applyTemplate template
  where
    sortedCtx = applySorting sorting ctxWOrule nodes
    ctxWOrule = ctxSetRule Nothing $ ctx
    nodes     = applySelect expr ctx

applyChoose :: Template -> Context -> [XmlTree]
applyChoose (TemplChoose whenList) ctx = applyWhenList whenList ctx

applyWhenList :: [When] -> Context -> [XmlTree]
applyWhenList []  _ = []
applyWhenList ((WhenPart expr template):xs) ctx =
  if applyTest expr ctx
    then applyTemplate template ctx
    else applyWhenList xs ctx

applyMessage :: Template -> Context -> [XmlTree]
applyMessage (TemplMessage halt template) ctx =
  if halt then error $ "Message(fatal): " ++ msg
          else trace ("Message(trace): " ++ msg) []
    where msg     = xshow content
          content = applyTemplate template ctx

-- -----------------------------------

applyElement :: Template -> Context -> [XmlTree]
applyElement (TemplElement compQName uris attribSets template) ctx =
    return $ createElement name uris aliases fullcontent
  where
    aliases     = getAliases $ ctxGetStylesheet ctx
    name        = applyComputedQName compQName ctx
    fullcontent = applyAttribSets [] attribSets ctx ++ applyTemplate template ctx

-- create an element from a list of attributes followed by content
createElement :: QName -> UriMapping -> NSAliasing -> [XmlTree] -> XmlTree
createElement name uris aliases fullcontent =
    mkElement name (nsAttrs ++ distinctAttribs) content
  where
    nsAttrs            = uriMap2Attrs $ aliasUriMapping aliases uris
    distinctAttribs    = nubBy eqAttr $ reverse attribs
    (attribs, content) = span (isAttr) fullcontent
    eqAttr node1 node2 = equivQName (fromJust $ getAttrName node1) (fromJust $ getAttrName node2)

applyAttribute :: Template -> Context -> [XmlTree]
applyAttribute (TemplAttribute compQName template) ctx =
    return $ mkAttr qName content
  where
    qName   = applyComputedQName compQName ctx
    content = applyTemplate template ctx

applyText :: Template -> Context -> [XmlTree]
applyText (TemplText s) _ = [mkText s]

applyValueOf :: Template -> Context -> [XmlTree]
applyValueOf (TemplValueOf expr) ctx = [mkText $ applyStringExpr expr ctx]

applyComment :: Template -> Context -> [XmlTree]
applyComment (TemplComment content) ctx =
    return $ mkCmt $ format $ collectTextnodes $ applyTemplate content ctx
  where
    format ""           = ""               -- could probably move to hxt...?
    format "-"          = "- "
    format ('-':'-':xs) = '-':' ':format ('-':xs)
    format (x:xs)       = x:format xs

applyProcInstr :: Template -> Context -> [XmlTree]
applyProcInstr (TemplProcInstr nameExpr template) ctx =
    return $ mkXPiTree name $ format $ collectTextnodes $ applyTemplate template ctx
  where
    name      = applyStringExpr nameExpr ctx
    format ""           = ""                       -- format = replaceAll "?>" "? >"
    format ('?':'>':xs) = '?':' ':'>':format xs    -- could probably move to hxt...?
    format (x:xs)       = x:format xs

-- -----------------------------------

applyApplTempl :: Template -> Context -> [XmlTree]
applyApplTempl (TemplApply expr mode args sorting) ctx =
    applyMatchRulesToEntireContext params rules mode sortedCtx
  where
    params      = createParamSet args ctx
    sortedCtx   = applySorting sorting ctx nodes
    nodes       = maybe (getChildrenNT $ ctxGetNode ctx)
                        (flip applySelect ctx)
                        expr
    rules       = getMatchRules $ ctxGetStylesheet ctx
```

```haskell
applyImports :: Template -> Context -> [XmlTree]
applyImports (TemplApplyImports) ctx=
    applyMatchRules Map.empty rules mode ctx
  where
    rules    = getRuleImports currRule
    mode     = getRuleMode currRule
    currRule = maybe (error "apply-imports must not be called during for-each") id $ ctxGetRule ctx

applyCallTempl  :: Template -> Context -> [XmlTree]
applyCallTempl (TemplCall name args) ctx =
    instantiateRule applyTemplate params rule ctx
  where
    params      = createParamSet args ctx
    rule        = maybe errNoRule id $ Map.lookup name rules
    rules       = getNamedRules $ ctxGetStylesheet ctx
    errNoRule   = error $ "No rule with qualified name: " ++ show name

-- ------------------------------------

applyCopy :: Template -> Context -> [XmlTree]
applyCopy (TemplCopy attrsets template) ctx =

    -- Case 1: Root node => just use the content template
    if isRoot currNode
    then applyTemplate template ctx

    -- Case 2: Any other element-node
    else if isElem currNode
    then return $ createElement name Map.empty Map.empty fullcontent

    -- otherwise: Just return the current node as result
    else return currNode

  where
    currNode    = subtreeNT $ ctxGetNode ctx
    name        = fromJust $ getElemName currNode
    fullcontent = applyAttribSets [] attrsets ctx ++ applyTemplate template ctx

applyCopyOf :: Template -> Context -> [XmlTree]
applyCopyOf (TemplCopyOf expr) = concatMap (expandRoot) . xPValue2XmlTrees . evalXPathExpr expr
  where expandRoot node = if isRoot node then getChildren node else return node

-- ------------------------------------

applyTemplate :: Template -> Context -> [XmlTree]
applyTemplate t@(TemplComposite _)     = applyComposite t
applyTemplate t@(TemplMessage _ _)     = applyMessage t
applyTemplate t@(TemplForEach _ _ _)   = applyForEach t
applyTemplate t@(TemplChoose _)        = applyChoose t
applyTemplate t@(TemplElement _ _ _ _) = applyElement t
applyTemplate t@(TemplAttribute _ _)   = applyAttribute t
applyTemplate t@(TemplText _)          = applyText t
applyTemplate t@(TemplValueOf _)       = applyValueOf t
applyTemplate t@(TemplComment _)       = applyComment t
applyTemplate t@(TemplProcInstr _ _)   = applyProcInstr t
applyTemplate t@(TemplApply _ _ _ _)   = applyApplTempl t
applyTemplate t@(TemplApplyImports)    = applyImports t
applyTemplate t@(TemplCall _ _)        = applyCallTempl t
applyTemplate t@(TemplCopy _ _)        = applyCopy t
applyTemplate t@(TemplCopyOf _)        = applyCopyOf t
applyTemplate t@(TemplVariable v)      = trace ("Warning: Unreacheable variable: "
                                                 ++ show (getVarName v)) const []

-- ------------------------------------
-- "Main" :

applyStylesheetWParams :: Map ExName Expr -> CompiledStylesheet -> XmlTree -> [XmlTree]
applyStylesheetWParams inputParams cs@(CompStylesheet matchRules _ vars _ strips _) rawDoc =
    map fixupNS $ applyMatchRules Map.empty matchRules Nothing ctxRoot
  where
    ctxRoot   = Ctx docNode [docNode] 1 1 gloVars Map.empty cs Nothing
    gloVars   = Map.map (evalVariableWParamSet extParams ctxRoot) vars
    extParams = Map.map (flip evalXPathExpr ctxRoot) inputParams
    docNode   = ntree $ stripDocument strips rawDoc

applyStylesheet :: CompiledStylesheet -> XmlTree -> [XmlTree]
applyStylesheet  = applyStylesheetWParams Map.empty
```

```
-- ------------------------------------
-- calling named- and applying match-rules

applyMatchRulesToChildren :: ParamSet -> [MatchRule] -> (Maybe ExName) -> Context -> [XmlTree]
applyMatchRulesToChildren args rules mode ctx =
    applyMatchRulesToEntireContext args rules mode childCtx
  where
    childCtx = ctxSetNodes (getChildrenNT $ ctxGetNode ctx) ctx

applyMatchRulesToEntireContext :: ParamSet -> [MatchRule] -> Maybe ExName -> Context -> [XmlTree]
applyMatchRulesToEntireContext args rules mode ctx = processContext ctx (applyMatchRules args rules
mode)

applyMatchRules :: ParamSet -> [MatchRule] -> (Maybe ExName) -> Context -> [XmlTree]
applyMatchRules _    []           mode ctx = matchDefaultRules mode ctx
applyMatchRules args (rule:rules) mode ctx =
    maybe (applyMatchRules args rules mode ctx)
          id
          (applyMatchRule args rule mode ctx)

applyMatchRule :: ParamSet -> MatchRule -> Maybe ExName -> Context -> Maybe [XmlTree]
applyMatchRule args rule@(MatRule expr _ ruleMode _ _ _) mode ctx =
  if mode==ruleMode && applyMatch expr ctx
    then Just $ instantiateRule applyTemplate args rule $ ctxSetRule (Just rule) ctx
    else Nothing

-- instantiateRule can either be used on match- or on named-rules.
-- It receives and processes the parameters and instantiate the rule-body.
-- The first argument will always be applyTemplate.
-- However, calling applyTemplate dircetly brakes Haskell's type system.
instantiateRule :: Rule a => (Template -> Context -> [XmlTree]) -> ParamSet -> a -> Context ->
[XmlTree]
instantiateRule applyTemplate args rule ctx =
    applyTemplate (getRuleContent rule) ctxNew
  where
    ctxNew = processParameters (getRuleParams rule) args $ clearLocalVariables ctx

matchDefaultRules :: (Maybe ExName) -> Context -> [XmlTree]
matchDefaultRules mode ctx@(Ctx ctxNavNode _ _ _ _ _ stylesheet _) =

    -- rules for match="*|/"
    if isElem ctxNode
    then applyMatchRulesToChildren Map.empty rules mode ctx

    -- rule for match="text()"
    else if isText ctxNode
    then [ctxNode]

    -- rule for match="@*"
    else if isAttr ctxNode
    then [mkText $ collectTextnodes $ getChildren ctxNode]

    -- the glorious rest (PIs and comments):
    else []

  where
    rules   = getMatchRules stylesheet
    ctxNode = subtreeNT ctxNavNode

-- ------------------------------------

-- Variables and Parameters

-- Evaluate a xsl:variable or xsl:param element and add the newly
-- created local variable to the context
processLocalVariable :: Variable -> ParamSet -> Context -> Context
processLocalVariable var@(MkVar _ name _) arguments ctx =
    addVariableBinding name val ctx
  where
    val = evalVariableWParamSet arguments ctx var

processParameters :: [Variable] -> ParamSet -> Context -> Context
processParameters params arguments ctx = foldl (\c v -> processLocalVariable v arguments c) ctx
params

evalVariableWParamSet :: ParamSet -> Context -> Variable -> XPathValue
```

```haskell
evalVariableWParamSet ps ctx (MkVar isParam name exprVar) =
    if isParam
    then maybe resultFromVar id $ Map.lookup name ps
    else resultFromVar
  where
    resultFromVar = evalXPathExpr exprVar ctx

-- create a set of parameters (Names refering to XPath-values) from a set of Variable-placeholders
(unevaluated expressions)
createParamSet :: Map ExName Variable -> Context -> ParamSet
createParamSet wParamList ctx = Map.map (evalVariableWParamSet Map.empty ctx) wParamList

-- ------------------------------------
-- handling of imported attributes

applyAttribSets :: [ExName] -> UsedAttribSets -> Context -> [XmlTree]
applyAttribSets callstack (UsedAttribSets sets) ctx = concatMap (\name -> applyAllAttrSetForName
callstack name ctx) sets

applyAllAttrSetForName :: [ExName] -> ExName -> Context -> [XmlTree]
applyAllAttrSetForName callstack name ctx =

    if name `elem` callstack
    then error $ "Attribute-Set " ++ show name ++ " is recursively used." ++
                 concatMap (("\n  used in "++) . show) callstack

    else if isNothing attrset
    then error $ "No attribute set with name: " ++ show name

    else concatMap (flip (applyAttribSet (name:callstack)) ctx) $ fromJust attrset

  where
    attrset = Map.lookup name $ getAttributeSets $ ctxGetStylesheet ctx

applyAttribSet :: [ExName] -> AttributeSet -> Context -> [XmlTree]
applyAttribSet callstack (AttribSet _ usedSets content) ctx =
    applyAttribSets callstack usedSets ctx ++ applyTemplate content ctx

-- ------------------------------------
-- Sorting

applySorting :: [SortKey] -> Context -> [NavXmlTree] -> Context
applySorting [] ctx nodes = ctxSetNodes nodes ctx
applySorting sortKeys ctx nodes =
    ctxSetNodes resultOrder ctx
  where
    resultOrder          = snd $ unzip sortedKVs
    sortedKVs            = sortBy compKV keysWVals
    keysWVals            = zip keys nodes
    keys                 = map extract nodes
    (extrFs, cmpFs)      = unzip $ map (flip applySortKey ctx) sortKeys

    -- helper functions:
    extract node         = map ($ ctxSetNodes [node] ctx) extrFs
    compKV (k1,_) (k2,_) = compressOrds $ compares k1 k2
    compares             = zipWith3 (($) $) cmpFs
    compressOrds         = maybe EQ id . find (/=EQ)

type SortVal = Either Float String

applySortKey :: SortKey -> Context -> ( Context -> SortVal
                                      , SortVal -> SortVal -> Ordering)
applySortKey (SortK expr typeATV orderATV) ctx =

    if typ/="number" && typ/="text"
    then error $ "unsupported type in xsl:sort: " ++ typ

    else if ord/="ascending" && ord/="descending"
    then error $ "order in xsl:sort element must be ascending or descending. Found: " ++ ord

    else (extractFct, cmpFct)

  where

    isNumber       = typ == "number"
    isDesc         = ord == "descending"
    ord            = applyStringExpr orderATV ctx
    typ            = applyStringExpr typeATV ctx

    extractFct ctx = let val  = applyStringExpr expr ctx in
                       if isNumber
                         then Left $ readWDefault (-1.0 / 0.0) val
                         else Right val

    cmpFct a       = (if isDesc then invertOrd else id)
                     . if isNumber then cmpNumber a else cmpString a
    cmpNumber (Left n1)  (Left n2)  = compare n1 n2
    cmpString (Right s1) (Right s2) = compare (map toLower s1) (map toLower s2)

invertOrd :: Ordering -> Ordering
invertOrd EQ = EQ
invertOrd LT = GT
invertOrd GT = LT

-- -----------------------------------
-- Namespace FIXUP

fixupNS :: XmlTree -> XmlTree
fixupNS = compressNS . disambigNS

compressNS :: XmlTree -> XmlTree
compressNS =
    mapTreeCtx compressElem $ Map.fromAscList [("xml", xmlNamespace), ("xmlns", xmlnsNamespace)]

compressElem :: UriMapping -> XNode -> (UriMapping, XNode)
compressElem uris node
  | isElem node = (newUris, changeAttrl (filter $ isImportant) node)
  | otherwise   = (uris, node)
  where
    newUris       = uris `Map.union` getUriMap node
    isImportant n = not (isNsAttr n)
                      || not ((localPart $ fromJust $ getAttrName n) `Map.member` uris)

disambigNS :: XmlTree -> XmlTree
disambigNS =
    mapTreeCtx step $ Map.fromAscList [("xml", xmlNamespace), ("xmlns", xmlnsNamespace)]
  where
    step uris node
      | isElem node = let uris'                = uris `Map.union` getUriMap node
                          (newUris, newNode') = disambigElem uris' node in
                        (newUris, setUriMap newUris newNode')
      | otherwise   = (uris, node)

disambigElem :: UriMapping -> XNode -> (UriMapping, XNode)
disambigElem nsMap elem =
    (newNsMap, mkEmptyElement
                 (remapNsName newNsMap $ fromJust $ getElemName elem)
                 $ map (changeName $ remapNsName newNsMap) $ fromJust $ getAttrl elem )
  where
    newNsMap   = nsMap `Map.union` Map.fromAscList newTuples
    newTuples  = zip newPrefs $ nub newUris
    newUris    = filter (`notElem` oldUris) $ filter (not . null) $ map namespaceUri $ mapMaybe
getName
                   (elem : map getNode (fromJust $ getAttrl elem))
    newPrefs   = filter (`notElem` oldPrefs) ["ns" ++ show i | i <- [1..]]
    oldPrefs   = Map.keys nsMap
    oldUris    = Map.elems nsMap

remapNsName :: UriMapping -> QName -> QName
remapNsName nsMap name =

    if maybe (nsUri=="") (== nsUri) luUri
    then name

    else mkQName newPref (localPart name) nsUri

  where
    luUri   = Map.lookup (namePrefix name) nsMap
    newPref = head $ (++ (error $ "int. error: No prefix for " ++ nsUri))
                $ Map.keys $ Map.filter (==namespaceUri name) nsMap
    nsUri   = namespaceUri name
```

Qualified element and attribute names

```
module Names where

import Common

xsltPrefix = "xsl"
xsltUri    = "http://www.w3.org/1999/XSL/Transform"

mkXsltName :: String -> QName
mkXsltName name = mkQName xsltPrefix name xsltUri

mkXsltAttribName :: String -> QName
mkXsltAttribName name = mkQName "" name ""

-- XSLT-Element QNames
xsltTransform                = mkXsltName "transform"
xsltStylesheet               = mkXsltName "stylesheet"
xsltMessage                  = mkXsltName "message"
xsltForEach                  = mkXsltName "for-each"
xsltChoose                   = mkXsltName "choose"
xsltWhen                     = mkXsltName "when"
xsltOtherwise                = mkXsltName "otherwise"
xsltIf                       = mkXsltName "if"
xsltElement                  = mkXsltName "element"
xsltAttribute                = mkXsltName "attribute"
xsltText                     = mkXsltName "text"
xsltValueOf                  = mkXsltName "value-of"
xsltComment                  = mkXsltName "comment"
xsltProcInstr                = mkXsltName "processing-instruction"
xsltInclude                  = mkXsltName "include"
xsltImport                   = mkXsltName "import"
xsltTemplate                 = mkXsltName "template"
xsltApplyTemplates           = mkXsltName "apply-templates"
xsltApplyImports             = mkXsltName "apply-imports"
xsltCallTemplate             = mkXsltName "call-template"
xsltVariable                 = mkXsltName "variable"
xsltParam                    = mkXsltName "param"
xsltWithParam                = mkXsltName "with-param"
xsltAttributeSet             = mkXsltName "attribute-set"
xsltCopy                     = mkXsltName "copy"
xsltCopyOf                   = mkXsltName "copy-of"
xsltSort                     = mkXsltName "sort"
xsltStripSpace               = mkXsltName "strip-space"
xsltPreserveSpace            = mkXsltName "preserve-space"
xsltNamespaceAlias           = mkXsltName "namespace-alias"

-- XSLT-Attribute QNames
xsltTerminate                = mkXsltAttribName "terminate"
xsltSelect                   = mkXsltAttribName "select"
xsltTest                     = mkXsltAttribName "test"
xsltName                     = mkXsltAttribName "name"
xsltNamespace                = mkXsltAttribName "namespace"
xsltUseAttributeSets         = mkXsltAttribName "use-attribute-sets"
xsltHRef                     = mkXsltAttribName "href"
xsltMatch                    = mkXsltAttribName "match"
xsltPriority                 = mkXsltAttribName "priority"
xsltMode                     = mkXsltAttribName "mode"
xsltDataType                 = mkXsltAttribName "data-type"
xsltOrder                    = mkXsltAttribName "order"
xsltElements                 = mkXsltAttribName "elements"
xsltStylesheetPrefix         = mkXsltAttribName "stylesheet-prefix"
xsltResultPrefix             = mkXsltAttribName "result-prefix"
xsltVersion                  = mkXsltAttribName "version"
xsltExlcudeResultPrefixes    = mkXsltAttribName "exclude-result-prefixes"
xsltExtensionElementPrefixes = mkXsltAttribName "extension-element-prefixes"

-- XSLT-Attribute QNames for special Literal result element attributes
xsltUseAttributeSetsLRE         = mkXsltName "use-attribute-sets"
xsltVersionLRE                  = mkXsltName "version"
xsltExlcudeResultPrefixesLRE    = mkXsltName "exclude-result-prefixes"
xsltExtensionElementPrefixesLRE = mkXsltName "extension-element-prefixes"

-- xml:space attribute-name
xmlSpace                     = mkQName "xml" "space" xmlNamespace
```