# The Functional Programming Language XSLT - A proof through examples

Dimitre Novatchev (mailto:dnovatchev@yahoo.com)
November, 2001

Until now it was believed that although XSLT is based on functional programming ideas, it is not as yet a full functional programming language, as it lacks the ability to treat functions as a first-class data type. Based on numerous concrete XSLT implementations of some of the major functional programming design patterns, including some of the most generic list-processing and tree-processing functions, this article provides ample proof that XSLT is in fact a full-pledged functional programming language. The presented code forms the base of a first XSLT functional programming library. It is emphasized that a decision to include higher-order functions support in XPath 2.0 will make functional programming in XSLT even more straightforward, natural and convenient.

This article can be found online at:
http://www.topxml.com/xsl/articles/fp/

Using MSXML? Get the MSXML source code at:
http://www.topxml.com/downloads/files/fxsl-MS.zip

Using SAXON? Get the MSXML source code at:
http://www.topxml.com/downloads/files/fxsl-Saxon.zip

## Introduction

XSLT has turned out to be very different from the typical programming languages in use today. One question that's being asked frequently is: "*What kind of programming language is actually XSLT?*". Until now, the authoritative answer from some of the best specialists was that XSLT is a declarative (as opposed to imperative) but still not a functional programming (FP) language. Michael Kay notes in his article "What kind of language is XSLT" [3] :
"*although XSLT is based on functional programming ideas, it is not as yet a full functional programming language, as it lacks the ability to treat functions as a first-class data type*".

What has been done to bring XML and FP languages closer together? One notable direction is to add XML support to some well-known FP languages, such as the HaXML compiler and tools written in Haskell [6], the LambdaXML xml-based FP language and the XML support in Erlang [2].

However, nothing is seemingly being done in the other direction -- to add functional programming capabilities to XSLT. The citation in the start of this section reflects the unanimous, dominating understanding of XSLT as a declarative-only programming language, which does not belong to the family of FP languages, due to its lack of support for higher order functions.

The purpose of this article is to demonstrate beyond any doubt that XSLT **is** a functional programming language in its own right. To achieve this goal, a reference XSLT implementation is provided of major FP concepts and functions. This article closely follows the contents and layout of John Hughes' article "Why functional programming matters" [1], in which some most typical and useful FP patterns and functions are defined. This article provides an XSLT implementation of most of the functions defined in [1]. Therefore, the reader is encouraged to read [1] either prior, or side-by-side with the current article.

## Starting point

As a starting point, let's define some basic terminology and present examples of the use of higher-order functions in FP. Then, we'll take a first step towards our FP implementation in XSLT by defining the "template reference" data type and demonstrate a simple example of its use.

### Imperative programming

The imperative style of programming describes a system as evolving from an initial state through a series of state changes to a set of desired final states. A program consists of commands that change the state of the system. For example:

```
   y = y + 3
```

will bring the system into a new state, in which the variable y has a new value, which has been obtained by adding 3 to the value of y in the previous state of the system.

### Declarative programming

The declarative programming style specifies relationships between different variables, e.g. the equation

```
 z = y + 3
```

declares *z* to have a value of three more than the value of *y*. Variables, once declared, cannot change their value. Typically, there is no concept of *state*, *order of execution*, *memory*,... etc.

In XSLT, the declarative approach is used: e.g.:

```
<xsl:variable name="z" select= "$y + 3" />
```

is the XSLT version of the mathematical equation above.

### Functional programming

A *function* is a relationship between a set of *inputs* and an *output*. It can also be regarded as an operation, which when passed specific values as input produces a specific output.

A *functional program* is made up of a series of definitions of functions and other values [7].

The functional programming style builds upon the declarative programming style by adding the ability to treat functions as first-class objects -- that means among other things that functions can be passed as arguments to other functions. A function can also return another function as its result.

## Higher-order functions in FP languages

A function is *higher order* if it takes a function as an argument or returns a function as a result, or both [7].

Examples.

A classical example is the ***map*** function, which can be defined in Haskell in the following two ways:

```
map   f    xs     =  [ f  x  |   x  <-  xs
]                              (1)
or
map   f   [ ]     =  [ ]

    (2)
map   f  (x:xs)   =   f  x   :   map   f    xs
```

The ***map*** function takes two arguments -- another function ***f*** and a list ***xs*** . The result is a list, every element of which is the result of applying ***f*** to the corresponding element of the list ***xs*** .

If we define **f** as :

```
f x = x + 3
```

and **xs** as

```
[1, 2, 3]
```

Then the value of

```
map   f    xs
```

is

```
[4, 5, 6]
```

The **map** function can be used to produce many other functions. For example, we can define a function to produce from a list a new one, whose elements' values are twice as big as the values of the elements of the input list:

```
doubleall  xs  =  map (* 2) xs
```

where (* 2) is a function, that produces a result twice as bigger as its input.

Another classical example is *functional composition*:

```
(.) f g x  =  f (g x)
```

## Higher-order functions in XSLT -- implementation of template references

Higher order functions can be implemented in XSLT by way of the *template reference* datatype.

### Explanation through an example

Templates in XSLT correspond to functions in FP languages. Named templates are always called by a corresponding xsl:call-template instruction. Templates with a match attribute are usually selected for instantiation in a less direct way -- from all templates matching the node-set specified in an xsl:apply-templates instruction.

Parameters can be declared for templates using xsl:param children of xsl:template and actual values for the parameters can be specified using xsl:with-param children of either xsl:call-template or xsl:apply-templates.

The result of calling/applying a template is always a particular output. Therefore, an XSLT template fits precisely the definition of a function.

As noted before, it is believed that XSLT is not a full functional programming language, because functions cannot be passed as parameters to other functions -- not that XSLT lacks functions at all. We accept that XSLT templates serve the role of functions, and want to show that functions (templates) can be passed as parameters to other functions (templates).

Unfortunately, it is not possible to specify through a variable the name of a template to be called, e.g.:

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

because according to the W3 XSLT Spec. [8] the value of the name attribute can only be a Q-Name. A Q-Name is static and must be completely specified  -- it cannot be dynamically produced by the contents of a variable.

Another way to instantiate a template dynamically has been known for quite some time [9], but there's no evidence until now of using it in a systematic manner. Let's have a template, which matches only a single node belonging to a unique namespace. In the rest of the text we'll call such nodes "*nodes having a unique type*":

File: example1.xsl

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version = "1.0" >
 <xsl:template
      match="*[namespace-uri()='8B9C63F4-F4AB5D11-994A0001-B4CD626F']">
     <xsl:param name = "pX" />

     <xsl:value-of select = "2 * $pX" />
 </xsl:template>

 <xsl:template
      match="*[namespace-uri()='AB02AC1C-1C65B3FF-77C5FFFE-4B329DA1']">
     <xsl:param name = "pX" />

     <xsl:value-of select = "3 * $pX" />
 </xsl:template>
</xsl:stylesheet>
```

We have defined two templates, each matching only a node of a unique type. The first template produces the value of its input parameter **pX** multiplied by 2. The second template produces the value of its input parameter **pX** multiplied by 3.

Now we'll define in another stylesheet a template that accepts as parameters references to two other templates (*template references*), instantiates the two templates that are referenced by its template-reference parameters, passing as parameter to each instantiation its **pX** parameter, then as result it produces the sum of the outputs of these instantiated templates.

```
<xsl:stylesheet  xmlns:xsl=http://www.w3.org/1999/XSL/Transform
  xmlns:f1="8B9C63F4-F4AB5D11-994A0001-B4CD626F"
  xmlns:f2="AB02AC1C-1C65B3FF-77C5FFFE-4B329DA1" >

     <xsl:import href = "example1.xsl" />
     <xsl:output method = "text" />

     <f1:f1/>
     <f2:f2/>

     <xsl:template match = "/" >
         <xsl:variable name = "vFun1" select = "document('')/*/f1:*[1]" />
         <xsl:variable name = "vFun2" select = "document('')/*/f2:*[1]" />

         <xsl:call-template name = "mySum" >
             <xsl:with-param name = "pX" select = "3" />
             <xsl:with-param name = "pFun1" select = "$vFun1" />
             <xsl:with-param name = "pFun2" select = "$vFun2" />
         </xsl:call-template>
     </xsl:template>

     <xsl:template name = "mySum" >
         <xsl:param name = "pX" />
         <xsl:param name = "pFun1" select = "/.." />
         <xsl:param name = "pFun2" select = "/.." />
```

```
        <xsl:variable name = ″vFx_1” >
            <xsl:apply-templates select = ″$pFun1” >
                <xsl:with-param name = ″pX” select = ″$pX” />
            </xsl:apply-templates>
        </xsl:variable>

        <xsl:variable name = ″vFx_2” >
            <xsl:apply-templates select = ″$pFun2” >
                <xsl:with-param name = ″pX” select = ″$pX” />
            </xsl:apply-templates>
        </xsl:variable>

        <xsl:value-of select = ″$vFx_1 + $vFx_2” />
    </xsl:template>

</xsl:stylesheet>
```

The result produced when this last stylesheet is applied on any (dummy) xml source document, is:

```
15
```

What we have effectively done is we called the template named "**mySum**", passing to it two *references to templates*, that accept a **pX** parameter and produce something out of it. The "**mySum**" template successfully instantiates (applies/calls) the templates that are uniquely identified by the template reference parameters, then produces the sum of their results. What guarantees that exactly the necessary templates will be selected by the XSLT processor is the unique namespace-uri of the nodes they are matching. The most important property of a template reference is that it guarantees the unique matching of the template that it is referencing.

The next two sections demonstrate that by using the template reference datatype just as described above we can implement even the most powerful FP design patterns/functions.

## Major FP design patterns/functions in XSLT

In this section we provide the XSLT implementation of many of the functions defined in John Hughes paper "Why functional programming matters" [1]. We provide numerous implementations of list-processing functions, and of a very generic tree-folding function and other functions that are based on it. We also demonstrate how *lazy evaluation* can be simulated in XSLT.

### Implemented Functions

The following table contains the names of all functions (in chronological order), whose XSLT implementation are provided as part of this article.

| | | |
|---|---|---|
| foldl | foldr | sum |
| product | sometrue | alltrue |
| minimum | maximum | append |
| map | doubleall | sumproducts |
| foldl-tree | sumTree | productTree |
| tree-labels-list | mapTree | buildListWhile |
| buildListUntil | take | takeWhile |
| drop | dropWhile | splitAt |
| span | partition | within |
| sqrt | buildListWhileMap | easyDiffList |
| elimError | improve | improvedDiff |
| partialSumsList | improvedIntegration | |

## List processing

The function **sum** that computes the sum of the elements of a list can be defined as follows:

```
sum []      =  0

sum (n:ns)  =  n + sum ns
```

The function **product** that computes the product of the elements of a list can be defined as follows:

```
product []      =  1

product (n:ns)  =  n  *  product ns
```

There is something common and general in the above two function definitions -- they define the same operation over a list, but provide different arguments to this operation. The arguments to the general list operation are displayed in bold above.

They are: a function **f** (+ and * in the described cases) that takes two arguments and an initial value (0 and 1 in the described cases) to use as a second argument when applying this function to the first element of the list. Therefore, we can define this general operation on lists as a function:

```
foldl f z [] = z

foldl f z (x:xs) = foldl f (f z x) xs
```

foldl processes a list from left to right. Its dual function, which processes a list from right to left is **foldr**:

```
foldr f z [] = z

foldr f z (x:xs) = f x (foldr f z xs)
```

We can define many functions just by feeding foldl (or foldr) with appropriate functions and null elements:

```
sum = foldl add 0

product = foldl multiply 1

sometrue = foldl or false

alltrue = foldl and true

maximum = foldl1 max

minimum = foldl1 min
```

where `foldl1` is defined as foldl which operates on non-empty lists, and `min(a1, a2)` is the lesser, while `max(a1, a2)` is the bigger of a couple of values.

```
append as bs = foldr (:) bs as

map f = foldl ((:).f ) [ ]
```

where **`(:)`** is the function, which adds an element to a list. Here's the corresponding XSLT implementation of **`foldl`**, **`foldr`**, and some of their useful applications:

**foldl:**

```
<xsl:template name = "foldl" >
    <xsl:param name = "pFunc" select = "/.." />
    <xsl:param name = "pA0" />
    <xsl:param name = "pList" select = "/.." />

    <xsl:choose>
        <xsl:when test = "not($pList)" >
            <xsl:copy-of select = "$pA0" />
        </xsl:when>

        <xsl:otherwise>
            <xsl:variable name = "vFunResult" >
                <xsl:apply-templates select = "$pFunc[1]" >
                    <xsl:with-param name = "arg0"
                                    select = "$pFunc[position() > 1]" />
                    <xsl:with-param name = "arg1" select = "$pA0" />
                    <xsl:with-param name = "arg2" select = "$pList[1]" />
                </xsl:apply-templates>
            </xsl:variable>

            <xsl:call-template name = "foldl" >
                <xsl:with-param name = "pFunc" select = "$pFunc" />
                <xsl:with-param name = "pList"
                                select = "$pList[position() > 1]" />
                <xsl:with-param name = "pA0" select = "$vFunResult" />
            </xsl:call-template>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

foldr:

```
<xsl:template name = "foldr" >
    <xsl:param name = "pFunc" select = "/.." />
    <xsl:param name = "pA0" />
    <xsl:param name = "pList" select = "/.." />

    <xsl:choose>
        <xsl:when test = "not($pList)" >
            <xsl:copy-of select = "$pA0" />
        </xsl:when>

        <xsl:otherwise>
            <xsl:variable name = "vFunResult" >
                <xsl:apply-templates select = "$pFunc[1]" >
                    <xsl:with-param name = "arg0"
                                    select = "$pFunc[position() > 1]" />
                    <xsl:with-param name = "arg1"
                                    select = "$ $pList[last()]" />
                    <xsl:with-param name = "arg2" select = "$ pA0" />
                </xsl:apply-templates>
            </xsl:variable>
            <xsl:call-template name = "foldr" >
                <xsl:with-param name = "pFunc" select = "$pFunc" />
                <xsl:with-param name = "pList"
                                select = "$pList[position() &lt; last()]" />
                <xsl:with-param name = "pA0" select = "$vFunResult" />
            </xsl:call-template>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

sum:

```
<xsl:stylesheet version = "1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sum-fold-func="sum-fold-func"
  exclude-result-prefixes = "xsl sum-fold-func" >

    <xsl:import href = "foldl.xsl" />

    <sum-fold-func:sum-fold-func/>

    <xsl:template name = "sum" >
        <xsl:param name = "pList" select = "/.." />

        <xsl:variable name = "sum-fold-func:vFoldFun"
                      select = "document('')/*/sum-fold-func:*[1]" />

        <xsl:call-template name = "foldl" >
            <xsl:with-param name = "pFunc"
                            select = "$sum-fold-func:vFoldFun" />
            <xsl:with-param name = "pList" select = "$pList" />
            <xsl:with-param name = "pA0" select = "0" />
        </xsl:call-template>
    </xsl:template>

    <xsl:template name = "add"
                  match = "*[namespace-uri() = 'sum-fold-func']" >
        <xsl:param name = "arg1" select = "0" />
        <xsl:param name = "arg2" select = "0" />

        <xsl:value-of select = "$arg1 + $arg2" />
    </xsl:template>

</xsl:stylesheet>
```

product:

```
<xsl:stylesheet version = "1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:prod-fold-func="prod-fold-func"
  exclude-result-prefixes = "xsl prod-fold-func" >

    <xsl:import href = "foldl.xsl" />

    <prod-fold-func:prod-fold-func/>

    <xsl:template name = "product" >
        <xsl:param name = "pList" select = "/.." />

        <xsl:variable name = "prod-fold-func:vFoldFun"
                      select = "document('')/*/prod-fold-func:*[1]" />

        <xsl:call-template name = "foldl" >
            <xsl:with-param name = "pFunc"
                            select = "$prod-fold-func:vFoldFun" />
            <xsl:with-param name = "pList" select = "$pList" />
            <xsl:with-param name = "pA0" select = "1" />
        </xsl:call-template>
    </xsl:template>

    <xsl:template name = "multiply"
                  match = "*[namespace-uri() = 'prod-fold-func']" >
        <xsl:param name = "arg1" select = "0" />
        <xsl:param name = "arg2" select = "0" />

        <xsl:value-of select = "$arg1 * $arg2" />
    </xsl:template>

</xsl:stylesheet>
```

sometrue:

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

    <xsl:import href = ”foldr.xsl” />


    <someTrue-Or:someTrue-Or/>

    <xsl:variable name = ”vOr”
                  select = ”document('')/*/someTrue-Or:*[1]” />

    <xsl:template name = ”someTrue” >
        <xsl:param name = ”pList” select = ”/..” />

        <xsl:call-template name = ”foldr” >
            <xsl:with-param name = ”pFunc” select = ”$vOr” />
            <xsl:with-param name = ”pA0” select = ”''“ />
            <xsl:with-param name = ”pList” select = ”$pList” />
        </xsl:call-template>
    </xsl:template>

    <xsl:template name = ”Or”
                  match = ”*[namespace-uri()='someTrue-Or']” >
        <xsl:param name = ”arg1” />
        <xsl:param name = ”arg2” />

        <xsl:if test = ”$arg1/node() or string($arg2)” >1</xsl:if>
    </xsl:template>

</xsl:stylesheet>
```

alltrue:

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

    <xsl:import href = ″foldr.xsl” />

    <allTrue-And:allTrue-And/>

    <xsl:template name = ″allTrue” >
        <xsl:param name = ″pList” select = ″/..” />

        <xsl:variable name = ″vAnd”
                      select = ″document('')/*/allTrue-And:*[1]” />

        <xsl:call-template name = ″foldr” >
            <xsl:with-param name = ″pFunc” select = ″$vAnd” />
            <xsl:with-param name = ″pA0” select = ″1” />
            <xsl:with-param name = ″pList” select = ″$pList” />
        </xsl:call-template>
    </xsl:template>

    <xsl:template name = ″And”
                  match = ″*[namespace-uri()='allTrue-And']” >
        <xsl:param name = ″arg1” />
        <xsl:param name = ″arg2” />

        <xsl:if test = ″$arg1/node() and string($arg2)” >1</xsl:if>
    </xsl:template>

</xsl:stylesheet>
```

minimum / maximum:

```
<xsl:stylesheet version = "1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:minimum-fold-func="minimum-fold-func"
  xmlns:minimum-pick-smaller="minimum-pick-smaller"
  xmlns:minimum-own-compare="minimum-own-compare"
  exclude-result-prefixes = "xsl minimum-fold-func
  minimum-own-compare minimum-pick-smaller" >

    <xsl:import href = "foldl.xsl" />

    <minimum-fold-func:minimum-fold-func/>
    <minimum-pick-smaller:minimum-pick-smaller/>
    <minimum-own-compare:minimum-own-compare/>

    <xsl:template name = "minimum" >
        <xsl:param name = "pList" select = "/.." />
        <xsl:param name = "pCMPFun" select = "/.." />

        <xsl:variable name = "vdfCMPFun"
                      select = "document('')/*/minimum-own-compare:*[1]" />

        <xsl:variable name = "vFoldFun"
                      select = "document('')/*/minimum-pick-smaller:*[1]" />

        <xsl:if test = "$pList" >
            <xsl:variable name = "vCMPFun"
                          select= "$pCMPFun | $vdfCMPFun[not($pCMPFun)]" />

            <xsl:variable name = "vFuncList" >
                <xsl:copy-of select = "$vFoldFun" /><!--Pick Smaller-->
                <xsl:copy-of select = "$vCMPFun" /> <!-- Compare -->
            </xsl:variable>

            <xsl:call-template name = "foldl" >
                <xsl:with-param name = "pFunc"
                                select = "msxsl:node-set($vFuncList)/*" />
                <xsl:with-param name = "pList" select = "$pList" />
                <xsl:with-param name = "pA0" select = "$pList[1]" />
            </xsl:call-template>
        </xsl:if>
    </xsl:template>

    <xsl:template name = "pickSmaller"
                  match = "*[namespace-uri() = 'minimum-pick-smaller']" >
        <xsl:param name = "arg0" />
        <xsl:param name = "arg1" />
        <xsl:param name = "arg2" />

        <xsl:variable name = "vIsSmaller" >
            <xsl:apply-templates select = "$arg0" >
                <xsl:with-param name = "arg1" select = "$arg1" />
                <xsl:with-param name = "arg2" select = "$arg2" />
            </xsl:apply-templates>
        </xsl:variable>
        <xsl:choose>
            <xsl:when test = "$vIsSmaller = 1" >
                <xsl:copy-of select = "$arg1" />
            </xsl:when>
            <xsl:otherwise>
                <xsl:copy-of select = "$arg2" />
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

     <xsl:template name = "isSmallerDefault"
                   match = "*[namespace-uri() = 'minimum-own-compare']" >
        <xsl:param name = "arg1" />
        <xsl:param name = "arg2" />

        <xsl:choose>
            <xsl:when test = "$arg1 < $arg2" >1</xsl:when>
            <xsl:otherwise>0</xsl:otherwise>
        </xsl:choose>
    </xsl:template>

</xsl:stylesheet>
```

append:

```
<xsl:stylesheet version = "1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:append-foldr-func="append-foldr-func"
  exclude-result-prefixes = "xsl append-foldr-func" >

    <xsl:import href = "foldr.xsl" />

    <append-foldr-func:append-foldr-func/>

    <xsl:template name = "append" >
        <xsl:param name = "pList1" select = "/.." />
        <xsl:param name = "pList2" select = "/.." />

        <xsl:variable name = "vFoldrFun"
                      select = "document('')/*/append-foldr-func:*[1]" />

        <xsl:call-template name = "foldr" >
            <xsl:with-param name = "pFunc" select = "$vFoldrFun" />
            <xsl:with-param name = "pList" select = "$pList1" />
            <xsl:with-param name = "pA0" select = "$pList2" />
        </xsl:call-template>
    </xsl:template>

    <xsl:template name = "appendL"
                  match = "*[namespace-uri() = 'append-foldr-func']" >
        <xsl:param name = "arg1" select = "/.." />
        <xsl:param name = ″arg2” select = ″/..” />

        <xsl:copy-of select = ″$arg1” />
        <xsl:copy-of select = ″$arg2” />
    </xsl:template>

</xsl:stylesheet>
```

map:

```
<xsl:stylesheet version = ″1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:map-foldl-func="map-foldl-func"
  exclude-result-prefixes = ″xsl map-foldl-func” >

    <xsl:import href = ″foldl.xsl” />

    <map-foldl-func:map-foldl-func/>

    <xsl:template name = ″map” >
        <xsl:param name = ″pFun” select = ″/..” />
        <xsl:param name = ″pList1” select = ″/..” />
        <xsl:param name = ″pList0” select = ″/..” />

        <xsl:variable name = ″vFoldlFun”
                      select = ″document('')/*/map-foldl-func:*[1]” />

        <xsl:variable name = ″vFuncComposition” >
            <xsl:copy-of select = ″$vFoldlFun” />
            <xsl:copy-of select = ″$pFun” />
        </xsl:variable>

        <xsl:variable name = ″vFComposition”
                      select = ″msxsl:node-set($vFuncComposition)/*” />

        <xsl:call-template name = ″foldl” >
            <xsl:with-param name=″pFunc” select = ″$vFComposition” />
            <xsl:with-param name = ″pList” select = ″$pList1” />
            <xsl:with-param name = ″pA0” select = ″/..” />
        </xsl:call-template>
    </xsl:template>

    <xsl:template name = ″mapL”
                  match = ″*[namespace-uri() = 'map-foldl-func']” >
        <xsl:param name = ″arg0” select = ″/..” />
        <xsl:param name = ″arg1” select = ″/..” />
        <xsl:param name = ″arg2” select = ″/..” />
         <!-- $arg1 must be A0 -->
        <xsl:copy-of select = ″$arg1” />
        <xsl:variable name = ″vFun1Result” >
           <xsl:apply-templates select = ″$arg0[1]” >
               <xsl:with-param name = ″arg1” select = ″$arg2[1]” />
           </xsl:apply-templates>
        </xsl:variable>

        <xsl:apply-templates select = ″$arg2” mode = ″copy” >
            <xsl:with-param name = ″pContents”
                            select = ″msxsl:node-set($vFun1Result)” />
        </xsl:apply-templates>
    </xsl:template>

    <xsl:template match = ″*” mode = ″copy” >
        <xsl:param name = ″pContents” />

         <xsl:copy>
            <xsl:copy-of select = ″$pContents” />
         </xsl:copy>
    </xsl:template>

</xsl:stylesheet>
```

doubleall

```
<xsl:stylesheet version = ″1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  exclude-result-prefixes = ″xsl doubleall” >

    <xsl:import href = ″map.xsl” />

    <doubleall:doubleall/>

    <xsl:template name = ″doubleall” >
        <xsl:param name = ″pList” select = ″/..” />

        <xsl:variable name = ″vFunDouble”
                      select = ″document('')/*/doubleall:*[1]” />

        <xsl:call-template name = ″map” >
            <xsl:with-param name = ″pFun” select = ″$vFunDouble” />
            <xsl:with-param name = ″pList1” select = ″$pList” />
        </xsl:call-template>
    </xsl:template>

    <xsl:template name = ″double”
                  match = ″*[namespace-uri() = 'doubleall']” >
        <xsl:param name = ″arg1” />

        <xsl:value-of select = ″2 * $arg1” />
    </xsl:template>

</xsl:stylesheet>
```

sumproducts.
This is a demonstration of the power of using **`map`** -- we will produce the sum of the products of all children of any /sales/sale element from the xml document as specified bellow:

xml document:

```
<sales>
    <sale>
        <price>3.5</price>
        <quantity>2</quantity>
        <Discount>0.75</Discount>
        <Discount>0.80</Discount>
        <Discount>0.90</Discount>
    </sale>
    <sale>
        <price>3.5</price>
        <quantity>2</quantity>
        <Discount>0.75</Discount>
        <Discount>0.80</Discount>
        <Discount>0.90</Discount>
    </sale>
</sales>
```

stylesheet:

```
<xsl:stylesheet version = ″1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:test-map-product="test-map-product"
  exclude-result-prefixes = ″xsl test-map-product” >

    <xsl:import href = ″sum.xsl” />
    <xsl:import href = ″map.xsl” />
    <xsl:import href = ″product.xsl” />

    <test-map-product:test-map-product/>

    <xsl:output method = ″text” />

    <xsl:template match = ″/” >
        <!-- Get: map product /sales/sale -->
        <xsl:variable name = ″vSalesTotals” >

            <xsl:variable name = ″vTestMap”
                          select =″document(")/*/test-map-product:*[1]” />

            <xsl:call-template name = ″map” >
                <xsl:with-param name = ″pFun” select = ″$vTestMap” />
                <xsl:with-param name = ″pList1” select = ″/sales/sale” />
            </xsl:call-template>
        </xsl:variable>
        <!-- Get sum map product /sales/sale -->
        <xsl:call-template name = ”sum” >
            <xsl:with-param name = ”pList”
                                select = ”msxsl:node-set($vSalesTotals)/*” />
        </xsl:call-template>
    </xsl:template>

    <xsl:template name = ”makeproduct”
                  match = ”*[namespace-uri() = 'test-map-product']” >
        <xsl:param name = ”arg1” />

        <xsl:call-template name = ”product” >
            <xsl:with-param name = ”pList” select = ”$arg1/*” />
        </xsl:call-template>
    </xsl:template>

</xsl:stylesheet>
```

Result:

```
7.5600000000000005
```

## Tree processing

The same generalised functions exist not only for lists, but also for trees. A tree is defined as a set of **node** (root) having a **label**X, and a set of sub-trees:

```
treeof X   ::  node X  [ (treeof X) ]
```

This effectively defines a datatype of ordered labeled trees, saying that a tree of Xs is a node, with a label which is an X, and a list of subtrees which are also trees of Xs [1].

Then we can define **foldl-tree** as follows:

```
foldl-tree  f  g  a  ( node, label, subtrees)  =

            f  label  ( foldl-tree_  f  g  a  subtrees )

foldl-tree_  f  g  a  ( subtree  restSubtrees ) =

  g (foldl-tree f g a  subtree) ( foldl-tree_  f  g  a  restSubtrees)
```

Probably, a better name for the above function would be **fold-top-bottom**.

As in the case of the list folding, the tree-folding function can produce a variety of useful functions on trees:

```
sumtree          =  foldl-tree  add  add  0

producttree      =  foldl-tree  multiply  multiply  1

tree-labels-list  =  foldl-tree  (:)  append  [ ]

maptree  f       =  foldl-tree  (makeNode . f) (:) [ ]
```

Bellow is the corresponding XSLT implementation:

foldl-tree:

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

       <xsl:template name = ″foldl-tree” >
           <xsl:param name = ″pFuncNode” select = ″/..” />
           <xsl:param name = ″pFuncSubtrees” select = ″/..” />
           <xsl:param name = ″pA0” />
           <xsl:param name = ″pNode” select = ″/..” />

           <xsl:choose>
               <xsl:when test = ″not($pNode)” >
                   <xsl:copy-of select = ″$pA0” />
               </xsl:when>
               <xsl:otherwise>
                   <xsl:variable name = ″vSubtrees”
                                 select = ″$pNode/*” />

                   <xsl:variable name = ″vSubTreeResult” >
                       <xsl:call-template name = ″foldl-tree_” >
                           <xsl:with-param name = ″pFuncNode”
                                           select = ″$pFuncNode” />
                           <xsl:with-param name = ″pFuncSubtrees”
                                           select= ″$pFuncSubtrees” />
                           <xsl:with-param name = ″pA0”
                                           select = ″$pA0” />
                           <xsl:with-param name = ″pSubTrees”
                                           select = ″$vSubtrees” />
                       </xsl:call-template>
                   </xsl:variable>

                   <xsl:apply-templates select = ″$pFuncNode[1]” >
                       <xsl:with-param name = ″arg0”
                                  select=″$pFuncNode[position() > 1]” />
                       <xsl:with-param name = ″arg1”
                                  select = ″$pNode/@tree-nodeLabel” />
                       <xsl:with-param name = ″arg2”
                                       select = ″$vSubTreeResult” />
                   </xsl:apply-templates>
               </xsl:otherwise>
           </xsl:choose>
       </xsl:template>

       <xsl:template name = ″foldl-tree_” >
           <xsl:param name = ″pFuncNode” select = ″/..” />
           <xsl:param name = ″pFuncSubtrees” select = ″/..” />
           <xsl:param name = ″pA0” />
           <xsl:param name = ″pSubTrees” select = ″/..” />
```

```
          <xsl:choose>
              <xsl:when test = ″not($pSubTrees)” >
                  <xsl:copy-of select = ″$pA0” />
              </xsl:when>
              <xsl:otherwise>
                  <xsl:variable name = ″vSubTree1Result” >
                      <xsl:call-template name = ″foldl-tree” >
                          <xsl:with-param name = ″pFuncNode”
                                          select = ″$pFuncNode” />
                          <xsl:with-param name = ″pFuncSubtrees”
                                          select= ″$pFuncSubtrees” />
                          <xsl:with-param name = ″pA0”
                                          select = ″$pA0” />
                          <xsl:with-param name = ″pNode”
                                          select = ″$pSubTrees[1]” />
                      </xsl:call-template>
                  </xsl:variable>

                  <xsl:variable name = ″vRestSubtreesResult” >
                      <xsl:call-template name = ″foldl-tree_” >
                          <xsl:with-param name = ″pFuncNode”
                                          select = ″$pFuncNode” />
                          <xsl:with-param name = ″pFuncSubtrees”
                                          select= ″$pFuncSubtrees” />
                          <xsl:with-param name = ″pA0”
                                          select = ″$pA0” />
                          <xsl:with-param name = ″pSubTrees”
                                   select= ″$pSubTrees[position() > 1]” />
                      </xsl:call-template>
                  </xsl:variable>

                  <xsl:apply-templates select = ″$pFuncSubtrees” >
                      <xsl:with-param name = ″arg0”
                              select = ″$pFuncSubtrees[position() > 1]” />
                      <xsl:with-param name = ″arg1”
                                      select = ″$vSubTree1Result” />
                      <xsl:with-param name = ″arg2”
                                      select=″$vRestSubtreesResult” />
                  </xsl:apply-templates>
              </xsl:otherwise>
          </xsl:choose>
      </xsl:template>

</xsl:stylesheet>
```

sumTree:

```
<xsl:stylesheet version = ″1.0”
 exclude-result-prefixes=“xsl add-tree”
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:add-tree="add-tree">

    <xsl:import href = ″foldl-tree.xsl” />

    <add-tree:add-tree/>

    <xsl:template name = ″sumTree” >
        <xsl:param name = ″pTree” />

        <xsl:variable name = ″vAdd”
                    select = ″document('')/*/add-tree:*[1]” />

        <xsl:call-template name = ″foldl-tree” >
            <xsl:with-param name = ″pFuncNode” select = ″$vAdd” />
            <xsl:with-param name = ″pFuncSubtrees” select = ″$vAdd” />
            <xsl:with-param name = ″pA0” select = ″0” />
            <xsl:with-param name = ″pNode” select = ″$pTree” />
        </xsl:call-template>
    </xsl:template>

    <xsl:template match = ″*[namespace-uri()='add-tree']” >
        <xsl:param name = ″arg1” />
        <xsl:param name = ″arg2” />

        <xsl:value-of select = ″$arg1 + $arg2” />
    </xsl:template>

</xsl:stylesheet>
```

producttree:

```
<xsl:stylesheet version = ”1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:mult-tree="mult-tree"
  exclude-result-prefixes = ”xsl mult-tree” >

    <xsl:import href = ”foldl-tree.xsl” />

    <mult-tree:mult-tree/>

    <xsl:template name = ”productTree” >
        <xsl:param name = ”pTree” />

        <xsl:variable name = ”vMult”
                            select = ”document('')/*/mult-tree:*[1]” />

        <xsl:call-template name = ”foldl-tree” >
            <xsl:with-param name = ”pFuncNode” select = ”$vMult” />
            <xsl:with-param name= ”pFuncSubtrees” select = ”$vMult” />
            <xsl:with-param name = ”pA0” select = ”1” />
            <xsl:with-param name = ”pNode” select = ”$pTree” />
        </xsl:call-template>
    </xsl:template>

    <xsl:template match = ”*[namespace-uri()='mult-tree']” >
        <xsl:param name = ”arg1” />
        <xsl:param name = ”arg2” />

        <xsl:value-of select = ”$arg1 * $arg2” />
    </xsl:template>

</xsl:stylesheet>
```

tree-labels-list:

```
<xsl:stylesheet version = ”1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:mult-tree="mult-tree"
  exclude-result-prefixes = ”xsl mult-tree” >

    <xsl:import href = ”foldl-tree.xsl” />

    <mult-tree:mult-tree/>

    <xsl:template name = ”productTree” >
        <xsl:param name = ”pTree” />

        <xsl:variable name = ”vMult”
                      select = ”document('')/*/mult-tree:*[1]” />

        <xsl:call-template name = ”foldl-tree” >
            <xsl:with-param name = ”pFuncNode” select = ”$vMult” />
            <xsl:with-param name= ”pFuncSubtrees” select = ”$vMult” />
            <xsl:with-param name = ”pA0” select = ”1” />
            <xsl:with-param name = ”pNode” select = ”$pTree” />
        </xsl:call-template>
    </xsl:template>

    <xsl:template match = ”*[namespace-uri()='mult-tree']” >
        <xsl:param name = ”arg1” />
        <xsl:param name = ”arg2” />

        <xsl:value-of select = ”$arg1 * $arg2” />
    </xsl:template>

</xsl:stylesheet>
```

mapTree:

```
<xsl:stylesheet version = "1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:tree-labels-append="tree-labels-append"
 xmlns:tree-labels-cons="tree-labels-cons"
 exclude-result-prefixes = "xsl msxsl mapTree-cons mapTree-makeNode-f">

    <xsl:import href = "foldl-tree.xsl" />

    <mapTree-cons:mapTree-cons/>
    <mapTree-makeNode-f:mapTree-makeNode-f/>

    <xsl:template name = "mapTree" >
        <xsl:param name = "pTree" />
        <xsl:param name = "pFun" />

        <xsl:variable name = "vmakeNode"
                      select= "document('')/*/mapTree-makeNode-f:*[1]" />

        <xsl:variable name = "vCons"
                      select = "document('')/*/mapTree-cons:*[1]" />

        <xsl:variable name = "vFuncNode" >
            <mapTree-makeNode-f:mapTree-makeNode-f/>
            <xsl:copy-of select = "$pFun" />
        </xsl:variable>

        <xsl:call-template name = "foldl-tree" >
            <xsl:with-param name = "pFuncNode"
                            select = "msxsl:node-set($vFuncNode)/*" />
            <xsl:with-param name= "pFuncSubtrees" select = "$vCons" />
            <xsl:with-param name = "pA0" select = "/.." />
            <xsl:with-param name = "pNode" select = "$pTree" />
        </xsl:call-template>
    </xsl:template>

    <xsl:template match = "*[namespace-uri()='mapTree-cons']" >
        <xsl:param name = "arg1" />
        <xsl:param name = "arg2" />

        <xsl:copy-of select = "$arg1" />
        <xsl:copy-of select = "$arg2" />

    </xsl:template>

    <xsl:template match = "*[namespace-uri()='mapTree-makeNode-f']" >
        <xsl:param name = "arg0" /> <!-- must contain pFun -->
        <xsl:param name = "arg1" /> <!-- must contain the node -->
        <xsl:param name = "arg2" /> <!-- must contan pA0 -->

        <xsl:variable name = "vFun1Result" >
            <xsl:apply-templates select = "$arg0[1]" >
```

```
                <xsl:with-param name = ”arg1” select = ”$arg1” />
            </xsl:apply-templates>
        </xsl:variable>

        <xsl:element name = ”{name($arg1/..)}” >
            <xsl:apply-templates select = ”$arg1” mode = ”copy” >
                <xsl:with-param name = ”pContents”
                                select=”msxsl:node-set($vFun1Result)” />
            </xsl:apply-templates>
            <xsl:copy-of select = ”$arg1/../node()[not(self::*)]” />
            <xsl:copy-of select = ”$arg2” />
        </xsl:element>
    </xsl:template>

    <xsl:template match = ”node()” mode = ”copy” >
        <xsl:param name = ”pContents” />

        <xsl:copy>
            <xsl:copy-of select = ”$pContents” />
        </xsl:copy>
    </xsl:template>

    <xsl:template match = ”@*” mode = ”copy” >
        <xsl:param name = ”pContents” />

        <xsl:attribute name = ”{name()}” >
            <xsl:copy-of select = ”$pContents” />
        </xsl:attribute>
    </xsl:template>

</xsl:stylesheet>
```

## Lazy evaluation

A very powerful feature of some FP languages (e.g. Haskell ) is having so called *non-strict functions* and *lazy evaluation*. A function **f** is said to be *strict* if, when applied to a non-terminating expression, it also fails to terminate. For most programming languages all functions are strict. But this is not so in Haskell. In Haskell if a function does not need a value from its argument, it never gets evaluated. Such kind of function evaluation is called *lazy evaluation*, and non-strict functions are called "lazy functions", and are said to evaluate their arguments "lazily", or "by need" [10]. With lazy evaluation it becomes possible to operate on *infinite data structures*. For example,

```
numsFrom  n     =   n : numsFrom  (n + 1)
```

evaluates to the infinite list of successive integers, beginning with n. From it, the infinite list of squares is constructed:

```
squares         =   map (^2)  (numsFrom 0)
```

With lazy evaluation implemented, a function can reference an infinite data structure and use only a finite number of its elements like in:

```
take 10 squares =>  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

which takes the first 10 elements from the infinite list squares, or:

```
takeWhile (< 100) squares =>  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

which takes from the infinite list squares the first elements, which are less than 100, etc.

Because XSLT does not support lazy evaluation, we provide implementation of several functions from the Haskell Prelude, which perform (simulated) lazy evaluation of infinite lists:

The function **take** produces a list of the starting n elements of a given list. Its definition in Haskell is as follows:

```
take :: Int -> [a] -> [a]

take 0 _  = []

take _ [] = []

take n (x:xs) | n>0 = x : take (n-1) xs

take _ _ = error "Prelude.take: negative argument”
```

The function takeWhile produces a list consisting of all elements beginning from the start of a given list, which satisfy a given predicate.

```
takeWhile :: (a -> Bool) -> [a] -> [a]

takeWhile p [] = []

takeWhile p (x:xs)

   | p x = x : takeWhile p xs

   | otherwise = []
```

and we also implement a group of other Haskell functions on lists, but our implementation is restricted only on finite lists:

drop -- takes all the elements of a list with the exception of the first n.

```
drop :: Int -> [a] -> [a]

drop 0 xs = xs

drop _ [] = []

drop n (_:xs) | n>0 = drop (n-1) xs

drop _ _ = error "Prelude.drop: negative argument”
```

dropWhile -- produces a list consisting of all the elements of a given list with the exception of the starting group of elements, which satisfy a given predicate.

```
dropWhile :: (a -> Bool) -> [a] -> [a]

dropWhile p [] = []

dropWhile p xs@(x:xs')

   | p x = dropWhile p xs'

   | otherwise = xs
```

splitAt -- produces two lists from a given list when split at the n-th element.

```
splitAt :: Int -> [a] -> ([a], [a])

splitAt 0 xs = ([],xs)

splitAt _ [] = ([],[])

splitAt n (x:xs) | n>0 = (x:xs',xs'')
      where (xs',xs'') = splitAt (n-1) xs

splitAt _ _ = error "Prelude.splitAt: negative argument”
```

span -- produces two lists from a given list, the first consisting of all starting elements that satisfy a given predicate, and the second, consisting of the rest of the elements of the given list.

```
span        :: (a -> Bool) -> [a] -> ([a],[a])

span p []   = ([],[])

span p xs@(x:xs')

   | p x       = (x:ys, zs)

   | otherwise = ([],xs)

                where (ys,zs) = span p xs'
```

partition -- produces two lists: the first consisting of all the elements of a given element, which satisfy a given predicate, and the second consisting of the rest of the elements of the given list.

```
partition :: (a -> Bool) -> [a] -> ([a],[a])

partition p xs = foldr select ([],[]) xs

              where select x (ts,fs) | p x = (x:ts,fs)

                                     | otherwise = (ts,x:fs)
```

The code of the implemented XSLT functions is presented below:

buildListWhile

```
<xsl:stylesheet version = ″1.0” exclude-result-prefixes = ″xsl msxsl”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt">

    <xsl:template name = ″buildListWhile” >
        <xsl:param name = ″pGenerator” select = ″/..” />
        <xsl:param name = ″pParamGenerator” select = ″/..” />
        <xsl:param name = ″pController” select = ″/..” />
        <xsl:param name = ″pParam0” select = ″/..” />
        <xsl:param name = ″pContollerParam” select = ″/..” />
        <xsl:param name = ″pElementName” select = ″'el'“ />
        <xsl:param name = ″pList” select = ″/..” />

        <xsl:if test = ″not($pController)” >
            <xsl:message terminate = ″yes” >
              [buildListWhile]Error: No pController specified:

              would cause infinite processing.
            </xsl:message>
        </xsl:if>

        <xsl:variable name = ″vElement” >
            <xsl:element name = ″{$pElementName}” >
                <xsl:apply-templates select = ″$pGenerator” >
                    <xsl:with-param name = ″pParams”
```

```
                                        select = ″$pParam0” />
                    <xsl:with-param name = ″pParamGenerator”
                                        select = ″$pParamGenerator” />
                    <xsl:with-param name = ″pList” select = ″$pList” />
                </xsl:apply-templates>
            </xsl:element>
        </xsl:variable>

        <xsl:variable name = ″newList” >
            <xsl:copy-of select = ″$pList” />
            <xsl:copy-of select = ″msxsl:node-set($vElement)/*” />
        </xsl:variable>

        <xsl:variable name = ″vResultList”
                      select = ″msxsl:node-set($newList)/*” />

        <xsl:variable name = ″vAccept” >
            <xsl:apply-templates select = ″$pController” >
                <xsl:with-param name = ″pList”
                                select = ″$vResultList” />
                <xsl:with-param name = ″pParams”
                                select = ″$pContollerParam” />
            </xsl:apply-templates>
        </xsl:variable>

        <xsl:choose>
            <xsl:when test = ″not(string($vAccept))” >
                <xsl:copy-of select = ″$pList” />
            </xsl:when>
            <xsl:otherwise>
                <xsl:call-template name = ″buildListWhile” >
                    <xsl:with-param name = ″pGenerator”
                                        select = ″$pGenerator” />
                    <xsl:with-param name = ″pParamGenerator”
                                        select = ″$pParamGenerator” />
                    <xsl:with-param name = ″pController”
                                        select = ″$pController” />
                    <xsl:with-param name = ″pContollerParam”
                                        select = ″$pContollerParam” />
                    <xsl:with-param name = ″pParam0”
                                        select = ″$pParam0” />
                    <xsl:with-param name = ″pElementName”
                                        select = ″$pElementName” />
                    <xsl:with-param name = ″pList”
                                        select = ″$vResultList” />
                </xsl:call-template>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

</xsl:stylesheet>
```

buildListUntil

```
<xsl:stylesheet version = ”1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  exclude-result-prefixes = ”xsl msxsl” >

    <xsl:template name = ”buildListUntil” >
        <xsl:param name = ”pGenerator” select = ”/..” />
        <xsl:param name = ”pController” select = ”/..” />
        <xsl:param name = ”pParam0” select = ”/..” />
        <xsl:param name = ”pParamGenerator” select = ”/..” />
        <xsl:param name = ”pElementName” select = ”'el'“ />
        <xsl:param name = ”pList” select = ”/..” />

        <xsl:if test = ”not($pController)” >
            <xsl:message terminate = ”yes” >
              [buildListUntil]Error:
              No pController specified: would cause infinite
              processing.
            </xsl:message>
        </xsl:if>

        <xsl:variable name = ”vElement” >
            <xsl:element name = ”{$pElementName}” >
                <xsl:apply-templates select = ”$pGenerator” >
                    <xsl:with-param name = ”pParams”
                                    select = ”$pParam0” />
                    <xsl:with-param name = ”pList” select = ”$pList” />
                </xsl:apply-templates>
            </xsl:element>
        </xsl:variable>

        <xsl:variable name = ”newList” >
            <xsl:copy-of select = ”$pList” />
            <xsl:copy-of select = ”msxsl:node-set($vElement)/*” />
        </xsl:variable>

        <xsl:variable name = ”vResultList”
                      select = ”msxsl:node-set($newList)/*” />

        <xsl:variable name = ”vShouldStop” >
            <xsl:apply-templates select = ”$pController” >
                <xsl:with-param name = ”pList” select= ”$vResultList” />
                <xsl:with-param name = ”pParams” select=”$pParam0” />
            </xsl:apply-templates>
        </xsl:variable>

        <xsl:choose>
            <xsl:when test = ”string($vShouldStop)” >
```

```
                <xsl:copy-of select = ″$vResultList” />
            </xsl:when>
            <xsl:otherwise>

                <xsl:variable name = ″vNewParams” >
                    <xsl:apply-templates select = ″$pParamGenerator” >
                        <xsl:with-param name = ″pList”
                                        select = ″$vResultList” />
                        <xsl:with-param name = ″pParams”
                                        select = ″$pParam0” />
                    </xsl:apply-templates>
                </xsl:variable>

                <xsl:call-template name = ″buildListUntil” >
                    <xsl:with-param name = ″pGenerator”
                                    select = ″$pGenerator” />
                    <xsl:with-param name = ″pController”
                                    select = ″$pController” />
                    <xsl:with-param name = ″pParam0”
                            select = ″msxsl:node-set($vNewParams)/*” />
                    <xsl:with-param name = ″pParamGenerator”
                                    select = ″$pParamGenerator” />
                    <xsl:with-param name = ″pElementName”
                                    select = ″$pElementName” />
                    <xsl:with-param name = ″pList”
                                    select = ″$vResultList” />
                </xsl:call-template>

            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

</xsl:stylesheet>
```

take

```
<xsl:stylesheet version = "1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:take-controller="take-controller"
  xmlns:take-dynamic-controller="take-dynamic-controller"
  exclude-result-prefixes = "xsl msxsl" >

    <xsl:import href = "buildListWhile.xsl" />

    <take-controller:take-controller/>

    <take-dynamic-controller:take-dynamic-controller/>

    <xsl:template name = "take" >
        <xsl:param name = "pList" select = "/.." />
        <xsl:param name = "pN" select = "0" />
        <xsl:param name = "pGenerator" select = "/.." />
        <xsl:param name = "pParam0" select = "/.." />
        <xsl:param name = "pParamGenerator" select = "/.." />
        <xsl:param name = "pElementName" select = "'el'" />

        <xsl:variable name = "vTakeController"
                           select = "document('')/*/take-controller:*[1]" />
        <xsl:variable name = "vTakeDynController"
                           select="document('')/*/take-dynamic-controller:*[1]" />

        <xsl:choose>
            <xsl:when test = "$pList" >
                <xsl:copy-of select = "$pList[position() <= $pN]" />
            </xsl:when>
            <xsl:when test = "$pGenerator" >
                <xsl:call-template name = "buildListWhile" >
                    <xsl:with-param name = "pList"
                                    select = "/.." />
                    <xsl:with-param name = "pGenerator"
                                    select = "$pGenerator" />
                    <xsl:with-param name = "pController"
                                   select = "$vTakeDynController" />
                    <xsl:with-param name = "pContollerParam"
                                    select = "$pN" />
                    <xsl:with-param name = "pParam0"
                                    select = "$pParam0" />
                    <xsl:with-param name = "pParamGenerator"
                                    select = "$pParamGenerator" />
                    <xsl:with-param name = "pElementName"
                                    select = "$pElementName" />
                </xsl:call-template>
            </xsl:when>
        </xsl:choose>
    </xsl:template>

    <xsl:template name = "takeController"
                  match = "*[namespace-uri()='take-controller']" >
        <xsl:param name = "pParams" />

        <xsl:if test = "$pParams > 0" >1</xsl:if>
    </xsl:template>

    <xsl:template name = "takeDynController"
                  match = "*[namespace-uri()='take-dynamic-controller']" >
        <xsl:param name = "pList" />
        <xsl:param name = "pParams" />

        <xsl:if test = "$pParams >= count($pList)" >1</xsl:if>
    </xsl:template>

</xsl:stylesheet>
```

takeWhile

```
<xsl:stylesheet version = "1.0"
  xmlns:xsl=http://www.w3.org/1999/XSL/Transform
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  exclude-result-prefixes = "xsl msxsl" >

    <xsl:import href = "buildListWhile.xsl" />

    <xsl:template name = "takeWhile" >
        <xsl:param name = "pList" select = "/.." />
        <xsl:param name = "pController" select = "/.." />
        <xsl:param name = "pContollerParam" select = "/.." />
        <xsl:param name = "pGenerator" select = "/.." />
        <xsl:param name = "pParam0" select = "/.." />
        <xsl:param name = "pParamGenerator" select = "/.." />
        <xsl:param name = "pElementName" select = "'el'" />

        <xsl:if test = "not($pController)" >
            <xsl:message terminate = "yes" >
            [takeWhile]Error: pController not specified.
            </xsl:message>
        </xsl:if>

        <xsl:choose>
            <xsl:when test = "$pList" >
                <xsl:variable name = "vAccept" >
                    <xsl:apply-templates select = "$pController" >
                        <xsl:with-param name = "pList"
                                        select = "$pList[1]" />
                        <xsl:with-param name = "pParams"
                                        select = "$pContollerParam" />
                    </xsl:apply-templates>
                </xsl:variable>

                <xsl:if test = "string($vAccept)" >
                    <xsl:copy-of select = "$pList[1]" />
                    <xsl:call-template name = "takeWhile" >
                        <xsl:with-param name = "pList"
                                        select = "$pList[position() > 1]" />
                        <xsl:with-param name = "pController"
                                        select = "$pController" />
                        <xsl:with-param name = "pContollerParam"
                                        select = "$pContollerParam" />
                        <xsl:with-param name = "pGenerator"
                                        select = "$pGenerator" />
                        <xsl:with-param name = "pParam0"
                                        select = "$pParam0" />
                        <xsl:with-param name = "pParamGenerator"
                                        select = "$pParamGenerator" />
                        <xsl:with-param name = "pElementName"
                                        select = "$pElementName" />
                    </xsl:call-template>
                </xsl:if>

            </xsl:when>
            <xsl:when test = "$pGenerator" >

                <xsl:call-template name = "buildListWhile" >
                    <xsl:with-param name = "pList"
                                    select = "/.." />
                    <xsl:with-param name = "pGenerator"
                                    select = "$pGenerator" />
                    <xsl:with-param name = "pController"
                                    select = "$pController" />
                    <xsl:with-param name = "pContollerParam"
                                    select = "$pContollerParam" />
                    <xsl:with-param name = "pParam0"
                                    select = "$pParam0" />
                    <xsl:with-param name = "pParamGenerator"
                                    select = "$pParamGenerator" />
                    <xsl:with-param name = "pElementName"
                                    select = "$pElementName" />
                </xsl:call-template>

            </xsl:when>
        </xsl:choose>
    </xsl:template>

</xsl:stylesheet>
```

Functions only implemented for finite lists:

drop

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

    <xsl:template name = ″drop” >
        <xsl:param name = ″pList” select = ″/..” />
        <xsl:param name = ″pN” select = ″0” />

        <xsl:copy-of select = ″$pList[position() > $pN]” />
    </xsl:template>

</xsl:stylesheet>
```

dropWhile

```
<xsl:stylesheet version = ″1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  exclude-result-prefixes = ″xsl msxsl” >

    <xsl:template name = ″dropWhile” >
        <xsl:param name = ″pList” select = ″/..” />
        <xsl:param name = ″pController” select = ″/..” />
        <xsl:param name = ″pContollerParam” select = ″/..” />
        <xsl:param name = ″pParam0” select = ″/..” />
        <xsl:param name = ″pElementName” select = ″'el'“ />

        <xsl:if test = ″not($pController)” >
            <xsl:message terminate = ″yes” >
              [dropWhile]Error: pController not specified.
            </xsl:message>
        </xsl:if>

        <xsl:if test = ″$pList” >
            <xsl:variable name = ″vDrop” >
                <xsl:apply-templates select = ″$pController” >
                    <xsl:with-param name= ″pList” select=″$pList[1]” />
                    <xsl:with-param name = ″pParams”
                                    select = ″$pContollerParam” />
                </xsl:apply-templates>
            </xsl:variable>

            <xsl:choose>
                <xsl:when test = ″string($vDrop)” >
                    <xsl:call-template name = ″dropWhile” >
                        <xsl:with-param name = ″pList”
```

```
                                               select="$pList[position() > 1]" />
                                   <xsl:with-param name = "pController"
                                                   select = "$pController" />
                                   <xsl:with-param name = "pContollerParam"
                                                   select = "$pContollerParam" />
                                   <xsl:with-param name = "pParam0"
                                                   select = "$pParam0" />
                                   <xsl:with-param name = "pElementName"
                                                   select = "$pElementName" />
                               </xsl:call-template>
                           </xsl:when>

                           <xsl:otherwise>
                               <xsl:copy-of select = "$pList" />
                           </xsl:otherwise>
                       </xsl:choose>
                   </xsl:if>
               </xsl:template>

</xsl:stylesheet>
```

split

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

    <xsl:template name = "split" >
        <xsl:param name = "pList" select = "/.." />
        <xsl:param name = "pN" select = "0" />
        <xsl:param name = "pElementName" select = "'list'" />

        <xsl:element name = "{$pElementName}" >
            <xsl:copy-of select = "$pList[position() <= $pN]" />
        </xsl:element>

        <xsl:element name = "{$pElementName}" >
            <xsl:copy-of select = "$pList[position() > $pN]" />
        </xsl:element>

    </xsl:template>

</xsl:stylesheet>
```

span

```
<xsl:stylesheet version = "1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt">

    <xsl:import href = "takeWhile.xsl" />

    <xsl:template name = "span" >
        <xsl:param name = "pList" select = "/.." />
        <xsl:param name = "pController" select = "/.." />
        <xsl:param name = "pContollerParam" select = "/.." />
        <xsl:param name = "pParam0" select = "/.." />
        <xsl:param name = "pElementName" select = "'list'" />

        <xsl:variable name = "vRTF-Positive" >
            <xsl:call-template name = "takeWhile" >
                <xsl:with-param name = "pList" select = "$pList" />
                <xsl:with-param name = "pController"
                                select = "$pController" />
                <xsl:with-param name = "pContollerParam"
                               select = "$pContollerParam" />
                <xsl:with-param name = "pParam0" select="$pParam0" />
            </xsl:call-template>
        </xsl:variable>

        <xsl:variable name = "vPositive"
                      select = "msxsl:node-set($vRTF-Positive)/*" />

        <xsl:element name = "{$pElementName}" >
            <xsl:copy-of select = "$vPositive" />
        </xsl:element>

        <xsl:element name = "{$pElementName}" >
            <xsl:copy-of select= "$pList[position() > count($vPositive)]" />
        </xsl:element>

    </xsl:template>

</xsl:stylesheet>
```

partition

```
<xsl:stylesheet version = ″1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt">

    <xsl:import href = ″map.xsl” />

    <xsl:template name = ″partition” >
        <xsl:param name = ″pList” select = ″/..” />
        <xsl:param name = ″pController” select = ″/..” />
        <xsl:param name = ″pContollerParam” select = ″/..” />
        <xsl:param name = ″pParam0” select = ″/..” />
        <xsl:param name = ″pElementName” select = ″'list'“ />

        <xsl:variable name = ″vHoldsListRTF” >
            <xsl:call-template name = ″map” >
                <xsl:with-param name = ″pFun” select= ″$pController” />
                <xsl:with-param name = ″pList1” select = ″$pList” />
            </xsl:call-template>
        </xsl:variable>

        <xsl:variable name = ″vHoldsList”
                      select = ″msxsl:node-set($vHoldsListRTF)/*” />

        <xsl:element name = ″{$pElementName}” >
            <xsl:for-each select = ″$vHoldsList” >
                <xsl:variable name = ″vPosition” select = ″position()” />
                <xsl:if test = ″string(.)” >
                    <xsl:copy-of select=″$pList[position()=$vPosition]” />
                </xsl:if>
            </xsl:for-each>
        </xsl:element>

        <xsl:element name = ″{$pElementName}” >
            <xsl:for-each select = ″$vHoldsList” >
                <xsl:variable name = ″vPosition” select = ″position()” />
                <xsl:if test = ″not(string(.))” >
                    <xsl:copy-of select=″$pList[position()=$vPosition]” />
                </xsl:if>
            </xsl:for-each>
        </xsl:element>
    </xsl:template>

</xsl:stylesheet>
```

## Advanced XSLT FP applications

In this section we provide the XSLT implementation of some advanced numerical algorithms by using essentially higher order functions for list processing and list generation. Essentially used in these implementations is a function **within** , which is passed a list-generator function and a tolerance as parameters. **within**  uses the **buildListWhile** function with the list-generator passed to it as parameter and with its own controller function, which triggers the stop condition, whenever the difference between the last two generated list elements becomes less than the tolerance parameter **eps**.

Here's the XSLT implementation of **within**:

```
<xsl:stylesheet version = ″1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:MyWithinEpsController="MyWithinEpsController"
  exclude-result-prefixes = ″xsl msxsl MyWithinEpsController” >

    <xsl:import href = ″buildListWhile.xsl” />

    <MyWithinEpsController:MyWithinEpsController/>

    <xsl:variable name = ″vMyWithinEpsController”
                  select = ″document('')/*/MyWithinEpsController:*[1]” />

   <xsl:template name = ″within” >
        <xsl:param name = ″pGenerator” select = ″/..” />
        <xsl:param name = ″pParam0” select = ″/..” />
        <xsl:param name = ″Eps” select = ″0.1” />

        <xsl:variable name = ″vResultList” >
            <xsl:call-template name = ″buildListWhile” >
                <xsl:with-param name = ″pGenerator”
                                select = ″$pGenerator” />
                <xsl:with-param name = ″pParam0” select=″$pParam0” />
                <xsl:with-param name = ″pController”
                                select = ″$vMyWithinEpsController” />
                <xsl:with-param name = ″pContollerParam”
                                select = ″$Eps” />
            </xsl:call-template>
        </xsl:variable>

        <xsl:value-of select = ″msxsl:node-set($vResultList)/*[last()]” />
    </xsl:template>

    <xsl:template name = ″MyWithinEpsController”
                  match = ″*[namespace-uri()='MyWithinEpsController']” >

        <xsl:param name = ″pList” select = ″/..” />
        <xsl:param name = "pParams" />

        <xsl:choose>
            <xsl:when test = "count($pList) < 2" >1</xsl:when>
            <xsl:when test = "count($pList) >= 2" >
                <xsl:variable name = "lastDiff"
                              select= "$pList[last()] - $pList[last() - 1]" />
                <xsl:if test = "not($lastDiff <= $pParams
                                and $lastDiff >= (0 - $pParams)
                                  )" >1</xsl:if>
            </xsl:when>
            <xsl:otherwise>1</xsl:otherwise>
        </xsl:choose>
    </xsl:template>

</xsl:stylesheet>
```

## Square root

Let's first consider the Newton-Raphson algorithm for finding square roots. The square root of a number **N** is computed starting from an initial approximation a0 and successively computing better ones using the rule:

```
a(n+1) = (a(n) + N/a(n)) / 2
```

This sequence in fact converges quite rapidly. We'll use a tolerance (eps) as a stop criterion -- the computation will yield a final result whenever two successive sequence elements differ by less than the tolerance.

Here's the XSLT implementation of the **sqrt** function:

```
<xsl:stylesheet version = "1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:msxsl="urn:schemas-microsoft-com:xslt"
   xmlns:MyRepeatGenerator="MyRepeatGenerator"
   xmlns:MyRepeatableFunction="MyRepeatableFunction"
   exclude-result-prefixes = "xsl msxsl MyRepeatGenerator
  MyRepeatableFunction" >

    <xsl:import href = "within.xsl" />

    <MyRepeatGenerator:MyRepeatGenerator/>

    <MyRepeatableFunction:MyRepeatableFunction/>

    <xsl:template name = "sqrt" >
        <xsl:param name = "N" />
        <xsl:param name = "Eps" select = "0.1" />

        <xsl:variable name = "vMyRepeat"
                      select = "document('')/*/MyRepeatGenerator:*[1]" />

        <xsl:variable name = "vMyFunction"
                      select = "document('')/*/MyRepeatableFunction:*[1]" />

        <xsl:variable name = "vrtfParams" >
            <param>
                <xsl:value-of select = "$N div 2" />
            </param>

            <xsl:copy-of select = "$vMyFunction" />

            <param>
                <xsl:value-of select = "$N" />
            </param>
        </xsl:variable>
        <xsl:call-template name = "within" >
            <xsl:with-param name= "pGenerator" select="$vMyRepeat" />
            <xsl:with-param name = "pParam0"
                            select = "msxsl:node-set($vrtfParams)/*" />
            <xsl:with-param name = "Eps" select = "$Eps" />
        </xsl:call-template>
    </xsl:template>

    <xsl:template name = "myRepeater"
                  match = "*[namespace-uri()='MyRepeatGenerator']" >
        <xsl:param name = "pList" select = "/.." />
        <xsl:param name = "pParams" />

        <xsl:choose>
            <xsl:when test = "not($pList)" >
                <xsl:copy-of select = "$pParams[1]/node()" />
            </xsl:when>
            <xsl:otherwise>
                <xsl:apply-templates select = "$pParams[2]" >
                    <xsl:with-param name = "X"
                                    select = "$pList[last()]" />
                    <xsl:with-param name = "N"
                                    select = "$pParams[3]/node()" />
                </xsl:apply-templates>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

    <xsl:template name = "myRptFn"
                  match = "*[namespace-uri()='MyRepeatableFunction']" >
        <xsl:param name = "X" />
        <xsl:param name = "N" />

        <xsl:value-of select = "($X + ($N div $X)) div 2" />
    </xsl:template>

</xsl:stylesheet>
```

## Numerical differentiation

Differentiating a function at a point yields the slope of the graph of the function at that point. Therefore, a rough first calculation of **f'(x)** will be obtained by the following function:

```
easydiff  f  x  h   =   (f(x+h) - f x) / h
```

This function will return a good approximation when the value of **h** is very small. However, when **h** is very small, f(x+h)and f x are very close together and the rounding error in subtracting them may hopelessly distort the result. The strategy taken here is to start with relatively big initial value of **h,** then to repeat the calculation of easydiff with smaller and smaller values of **h** and to stop this calculation at the first occurrence of two consecutive results within the desired tolerance. For convenience we'll be using **h/2** in each new step of these calculations.

First we must define a function which computes the sequence:

```
differentiate h0 f x  = map (easydiff f x) (repeat halve h0)

where halve x = x/2
```

where **h0** is the initial value for **h** and its successive new values are obtained by repeated halving.

From the sequence thus obtained, we can compute the derivative at any point **x** by simply applying the **within** function like this:

```
within  eps  (differentiate h0 f x)
```

It turns out that even this solution is not perfect because of the slow speed of convergence of the sequence. Some math [1] can help in showing that the values of the sequence contain an error term, which is proportional to a power of **h**. Simple calculations, which we are not reproducing here show that the correct answer A can be obtained by the following formula:

```
    a(i+1)*(2**n) - a(i)
A = --------------------
         2**n - 1
```

Because the error is only roughly proportional to a power of **h**, the above result is also approximate, but it is a much better approximation.

We can apply this to all successive pairs of approximations using the function

```
elimerror n (a:b:rest)
                   = ((b*(2**n)-a)/(2**n-1)):(elimerror n1 (b:rest)))
```

Thus we obtain another sequence with error "eliminated" (in reality just much reduced), which converges much more rapidly. The only problem is that we do not know the value for **n** and **n1**. Again, we are using a function for the estimation of **n**, from [1].

```
order (a:b:c:rest) = round(log2((a-c)/(b-c) - 1))
```

Then we can finally define the function, which takes a sequence (list) generated with progressively halving **h**, and improves it:

```
improve as = elimerror (order as) as
```

And the derivative of the function will then be computed more precisely by:

```
within eps (improve differentiate h0 f x)
```

We must note here that **improve** only works on sequences of approximations computed using a parameter **h**, which is halved before the computation of each approximation. Remarkably, the result of **improve** is again such a sequence! And this effectively allows us to "improve improve":

```
within eps (improve (improve (improve (differentiate h0 f x))))
```

One could even go further and define:

```
super s = map second (repeat improve s)
```

where

```
second (x, y) = y
```

This repeatedly constructs more and more improved sequences of approximations from one another, then constructs a final sequence, taking every second element (John Hughes states they are the best [1]) from each improved sequence.

Finally, from this super-sequence we get the first element within our defined tolerance:

```
within eps (super (differentiate h0 f x))
```

Can these functions be implemented in XSLT? The answer is "yes".

Because in the definition of **super** above an infinite list is passed as an argument to the **map** function, we need to implement one more XSLT function that simulates Haskell's lazy evaluation. The  buildListWhileMap function simulates two infinite lists -- a list of parameter values, the latest of which is calculated in the last iteration, and a list of "result" values, that are calculated from the list of parameter values. It is passed 3 functions as parameters -- a *generator* function, which produces the new list of parameters and the new list of results from the previous ones by using the *mapping* function, which generates a value from every element of the list of parameters. In this way the list of parameters and the list of results will grow infinitely and this is only avoided by having a *controller* function implement the check for a stop criterion.

Below is the code of the buildListWhileMap function followed by a test example that uses it to implement the differentiate function as defined above.

buildListWhileMap

```
<xsl:stylesheet version = ″1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  exclude-result-prefixes = ″xsl msxsl” >

    <xsl:template name = ″buildListWhileMap” >
        <xsl:param name = ″pGenerator” select = ″/..” />
        <xsl:param name = ″pParam0” select = ″/..” />
        <xsl:param name = ″pController” select = ″/..” />
        <xsl:param name = ″pContollerParam” select = ″/..” />
        <xsl:param name = ″pMap” select = ″/..” />
        <xsl:param name = ″pMapParams” select = ″/..” />
        <xsl:param name = ″pElementName” select = ″'el'“ />
        <xsl:param name = ″pList” select = ″/..” />
        <xsl:param name = ″pBaseList” select = ″/..” />

        <xsl:if test = ″not($pController)” >
            <xsl:message terminate = ″yes” >
               [buildListWhileMap]Error:
               No pController specified: would cause infinite
               processing.
            </xsl:message>
        </xsl:if>

        <xsl:variable name = ″vNewBaseListElement” >
            <xsl:if test = ″$pMap” >
                <xsl:element name = ″{$pElementName}” >
                    <xsl:apply-templates select = ″$pGenerator” >
                        <xsl:with-param name = ″pParams”
                                        select = ″$pParam0” />
                        <xsl:with-param name = ″pList”
                                        select = ″$pBaseList” />
                    </xsl:apply-templates>
                </xsl:element>
            </xsl:if>
        </xsl:variable>

        <xsl:variable name = ″vElement” >
            <xsl:element name = ″{$pElementName}” >
                <xsl:choose>
                    <xsl:when test = ″not($pMap)” >
                         <xsl:apply-templates select = ″$pGenerator” >
                            <xsl:with-param name = ″pParams”
                                            select = ″$pParam0” />
                            <xsl:with-param name = ″pList”
                                            select = ″$pList” />
                         </xsl:apply-templates>
                    </xsl:when>
                    <xsl:otherwise>

                        <xsl:apply-templates select = ″$pMap” >
                            <xsl:with-param name = ″pParams”
                                           select = ″$pMapParams” />
                            <xsl:with-param name = ″pDynParams”
                          select = ″msxsl:node-set($vNewBaseListElement)
                                                    / *” />
                            <xsl:with-param name = ″pList”
                                           select = ″$pList” />
                        </xsl:apply-templates>

                    </xsl:otherwise>
                </xsl:choose>
            </xsl:element>
        </xsl:variable>

        <xsl:variable name = ″newList” >
            <xsl:copy-of select = ″$pList” />
            <xsl:copy-of select = ″msxsl:node-set($vElement)/*” />
        </xsl:variable>

        <xsl:variable name = ″newRTFBaseList” >
            <xsl:copy-of select = ″$pBaseList” />
            <xsl:copy-of select=“msxsl:node-set($vNewBaseListElement)/*”
            />
        </xsl:variable>

        <xsl:variable name = ″vResultList”
                      select = ″msxsl:node-set($newList)/*” />

        <xsl:variable name = ″vResultBaseList”
                      select = ″msxsl:node-set($newRTFBaseList)/*” />

        <xsl:variable name = ″vAccept” >
            <xsl:apply-templates select = ″$pController” >
                <xsl:with-param name = ″pList” select= ″$vResultList” />
                <xsl:with-param name = ″pParams”
                                select = ″$pContollerParam” />
            </xsl:apply-templates>
        </xsl:variable>

        <xsl:choose>
            <xsl:when test = ″not(string($vAccept))” >
                <xsl:copy-of select = ″$pList” />
            </xsl:when>
            <xsl:otherwise>

                <xsl:call-template name = ″buildListWhileMap” >
                    <xsl:with-param name = ″pGenerator”
                                    select = ″$pGenerator” />
                    <xsl:with-param name = ”pParam0”
                                    select = ”$pParam0” />
                    <xsl:with-param name = ”pController”
                                    select = ”$pController” />
                    <xsl:with-param name = ”pContollerParam”
                                    select = ”$pContollerParam” />
                    <xsl:with-param name = ”pMap” select= ”$pMap” />
                    <xsl:with-param name = ”pMapParams”
                                    select = ”$pMapParams” />
                    <xsl:with-param name = ”pElementName”
                                    select = ”$pElementName” />
                    <xsl:with-param name = ”pList”
                                    select = ”$vResultList” />
                    <xsl:with-param name = ”pBaseList”
                                    select = ”$vResultBaseList” />
                </xsl:call-template>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

</xsl:stylesheet>
```

easyDiffList

```
<xsl:stylesheet version = ”1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:myHalve="myHalve"
  xmlns:myEasyDiffMap="myEasyDiffMap"
  xmlns:myWithinController="myWithinController"
  exclude-result-prefixes = ”xsl msxsl myHalve
                             myEasyDiffMap myWithinController” >

    <xsl:import href = ”buildListWhileMap.xsl” />

    <myHalve:myHalve/>

    <myWithinController:myWithinController/>

    <myEasyDiffMap:myEasyDiffMap/>

    <xsl:template name = ”easyDiffList” >
        <xsl:param name = ”pFun” select = ”/..” />
        <xsl:param name = ”pX” />
        <xsl:param name = ”pH0” select = ”0.1” />
        <xsl:param name = ”pEps” select = ”0.01” />

        <xsl:variable name = ”vMyHalveGenerator”
                      select = ”document('')/*/myHalve:*[1]” />

        <xsl:variable name = ”vmyWithinController”
                   select=”document('')/*/myWithinController:*[1]” />

        <xsl:variable name = ”vmyEasyDiffMap”
                      select = ”document('')/*/myEasyDiffMap:*[1]” />
        <xsl:variable name = ”fx” >
            <xsl:apply-templates select = ”$pFun” >
                <xsl:with-param name = ”pParam” select = ”$pX” />
            </xsl:apply-templates>
        </xsl:variable>

        <xsl:variable name = ”vrtfMapParams” >
            <xsl:copy-of select = ”$pFun” />
            <param>
                <xsl:value-of select = ”$pX” />
            </param>
            <param>
                <xsl:value-of select = ”$fx” />
            </param>
        </xsl:variable>

        <xsl:variable name = ”vMapParams”
                      select = ”msxsl:node-set($vrtfMapParams)/*” />

        <xsl:call-template name = ”buildListWhileMap” >
            <xsl:with-param name = ”pGenerator”
                            select = ”$vMyHalveGenerator” />
            <xsl:with-param name = ”pParam0” select = ”$pH0” />
            <xsl:with-param name = ”pController”
                            select = ”$vmyWithinController” />
            <xsl:with-param name= ”pContollerParam” select=”$pEps” />
            <xsl:with-param name = ”pMap” select=”$vmyEasyDiffMap” />
            <xsl:with-param name = ”pMapParams”
                            select = ”$vMapParams” />
        </xsl:call-template>
    </xsl:template>

    <xsl:template name = ”myHalveGenerator”
                  match = ”*[namespace-uri()='myHalve']” >
        <xsl:param name = ”pParams” select = ”/..” />
        <xsl:param name = ”pList” select = ”/..” />

        <xsl:choose>
            <xsl:when test = ”not($pList)” >
                <xsl:value-of select = ”$pParams” />
            </xsl:when>
            <xsl:otherwise>
                <xsl:value-of select = ”$pList[last()] div 2” />
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

    <xsl:template name = ”myWithinController”
                  match = ”*[namespace-uri()='myWithinController']” >
        <xsl:param name = ”pParams” select = ”/..” />
        <xsl:param name = ”pList” select = ”/..” />

        <xsl:variable name = ”vLastDiff”
                      select=”$pList[last()] - $pList[last() - 1]” />

        <xsl:choose>
            <xsl:when test=“not($vLastDiff < $pParams
                                and $vLastDiff > (0 - $pParams)
                                 )
```

```
                              and count($pList) <= 40 “ >1</xsl:when>
    </xsl:choose>
  </xsl:template>

  <xsl:template name = ”myEasyDiffMap”
                match = ”*[namespace-uri()='myEasyDiffMap']” >
    <xsl:param name = ”pParams” select = ”/..” />
    <xsl:param name = ”pDynParams” select = ”/..” />
    <xsl:param name = ”pList” select = ”/..” />

    <xsl:variable name = ”x” select = ”$pParams[2]” />

    <xsl:variable name = ”h” select = ”$pDynParams[1]” />

    <xsl:choose>
      <xsl:when test = ”not($h = 0)” >
        <xsl:variable name = ”fx_plus_h” >
          <xsl:apply-templates select = ”$pParams[1]” >
            <xsl:with-param name=”pParam” select=”$x + $h” />
          </xsl:apply-templates>
        </xsl:variable>

        <xsl:variable name = ”fx” >
          <xsl:choose>
            <xsl:when test = ”count($pParams) >= 3” >
              <xsl:value-of select = ”$pParams[3]” />
            </xsl:when>
            <xsl:otherwise>
              <xsl:apply-templates select = ”$pParams[1]” >
                <xsl:with-param name = ”pParam” select = ”$x” />
              </xsl:apply-templates>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:variable>

        <xsl:value-of select = ”($fx_plus_h - $fx) div $h” />
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select = ”$pList[last()]” />
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

</xsl:stylesheet>
```

Let's now define the `elimerror` function. The code bellow also contains an implementation for the auxiliary function `getOrder`, which calculates the order **`n`** that must be used in the elimination of errors formula. This is the only place, where we're using essentially an extension function in order to calculate `round(log2(x))`. In the current implementation a Javascript extension function is used.

elimerror

```
<xsl:stylesheet version = ″1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:roundLog2="roundLog2">

    <xsl:import href = ″roundLog2.xsl” />

    <xsl:template name = ″elimError” >
        <xsl:param name = ″pList” select = ″/..” />
        <xsl:param name = ″pOrder” select = ″-9999999” />
        <xsl:param name = ″pv2__N” select = ″0” />

        <xsl:choose>
            <xsl:when test = ″count($pList) < 3” >
                <xsl:copy-of select = ″$pList” />
            </xsl:when>
            <xsl:otherwise>

                <xsl:variable name = ″vOrder” >
                    <xsl:choose>
                        <xsl:when test = ″$pOrder < -9999998” >
                            <xsl:call-template name = ″getOrder” >
                                <xsl:with-param name = ″pList”
                                                select = ″$pList” />
                            </xsl:call-template>
                        </xsl:when>
                        <xsl:otherwise>
                            <xsl:value-of select = ″$pOrder” />
                        </xsl:otherwise>
                    </xsl:choose>
                </xsl:variable>

                <xsl:variable name = ″v2__N” >
                    <xsl:choose>
                        <xsl:when test = ″$pv2__N = 0” >
                            <xsl:call-template name = ″pow” >
                                <xsl:with-param name = ″base”
                                                select = ″2” />
                                <xsl:with-param name = ″pow”
                                                select = ″$vOrder” />
                            </xsl:call-template>
                        </xsl:when>
                        <xsl:otherwise>
                            <xsl:value-of select = ″$pv2__N” />
                        </xsl:otherwise>
                    </xsl:choose>
                </xsl:variable>

                <xsl:element name = ″{name($pList[1])}” >
                  <xsl:value-of select = ″($pList[2] * $v2__N - $pList[1])
                                       div ($v2__N - 1)” />
                </xsl:element>

                <xsl:variable name = ″vNewList”
                              select = ″$pList[position() > 1]” />
                <xsl:variable name = ”vNewOrder” >
                    <xsl:call-template name = ”getOrder” >
                        <xsl:with-param name = ”pList”
                                        select = ”$vNewList” />
                    </xsl:call-template>
                </xsl:variable>

                <xsl:call-template name = ”elimError” >
                    <xsl:with-param name = ”pList”
                                    select = ”$vNewList” />
                    <xsl:with-param name = ”pOrder”
                                    select = ”$vNewOrder” />
                    <xsl:with-param name = ”pv2__N”
                                    select = ”$v2__N” />
                </xsl:call-template>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

    <xsl:template name = ”getOrder” >
        <xsl:param name = ”pList” select = ”/..” />

        <xsl:choose>
            <xsl:when test = ”count($pList) < 3” >1</xsl:when>
            <xsl:otherwise>

                <xsl:variable name = ”v1”
                              select = ”($pList[1] - $pList[3])
                                        div ($pList[2] - $pList[3])
                                        - 1” />

                <xsl:variable name = ”v2” >

                    <xsl:choose>
                        <xsl:when test = ”$v1 > 0” >
                            <xsl:value-of select = ”$v1” />
                        </xsl:when>
                        <xsl:when test = ”$v1 < 0” >
                            <xsl:value-of select = ”(-1) * $v1” />
                        </xsl:when>
                        <xsl:otherwise>1</xsl:otherwise>
                    </xsl:choose>
                </xsl:variable>

                <xsl:value-of
                         select = ”roundLog2:roundLog2(string($v2))” />
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

    <xsl:template name = ”pow” >
        <xsl:param name = ”base” select = ”1” />
        <xsl:param name = ”pow” select = ”0” />
        <xsl:param name = ”tmpResult” select = ”1” />

        <xsl:variable name = ”result” >
            <xsl:choose>
                <xsl:when test = ”$pow >= 0” >
```

```
                    <xsl:value-of select = ″$tmpResult * $base” />
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select = ″$tmpResult div $base” />
                </xsl:otherwise>
            </xsl:choose>
        </xsl:variable>

        <xsl:variable name = ″incr” >
            <xsl:choose>
                <xsl:when test = ″$pow >= 0” >
                    <xsl:value-of select = ″- 1” />
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select = ″1” />
                </xsl:otherwise>
            </xsl:choose>
        </xsl:variable>

        <xsl:choose>
            <xsl:when test = ″$pow = 0” >
                <xsl:value-of select = ″$tmpResult” />
            </xsl:when>
            <xsl:otherwise>
                <xsl:call-template name = ″pow” >
                    <xsl:with-param name = ″base” select = ″$base” />
                    <xsl:with-param name= ″pow” select=″$pow + $incr” />
                    <xsl:with-param name = ″tmpResult” select=″$result” />
                </xsl:call-template>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

</xsl:stylesheet>
```

It is easy now to define the **`improve`** function.

improve

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

    <xsl:import href = ″elimError.xsl” />

    <xsl:template name = ″improve” >
        <xsl:param name = ″pList” select = ″/..” />

        <xsl:variable name = ″vOrder” >
            <xsl:call-template name = ″getOrder” >
                <xsl:with-param name = ″pList” select = ″$pList” />
            </xsl:call-template>
        </xsl:variable>

        <xsl:variable name = ″v1Order” >
            <xsl:choose>
                <xsl:when test = ″$vOrder = 0” >1</xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select = ″$vOrder” />
                </xsl:otherwise>
            </xsl:choose>
        </xsl:variable>

        <xsl:call-template name = ″elimError” >
            <xsl:with-param name = ″pList” select = ″$pList” />
            <xsl:with-param name = ″pOrder” select = ″$v1Order” />
        </xsl:call-template>
    </xsl:template>

</xsl:stylesheet>
```

Finally, the `improvedDiff` function uses improve to calculate the final improved derivative of a function **`f`** at point **`x`**. We are producing a list of successively improved lists of  values for **`f'`** and taking every second element of these lists, until our tolerance criterion has been satisfied.

improvedDiff

```
<xsl:stylesheet version = ”1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:myImproveGenerator="myImproveGenerator"
  xmlns:myImproveController="myImproveController"
  exclude-result-prefixes = ”xsl msxsl
  myImproveGenerator myImproveController” >

    <xsl:import href = ”improve.xsl” />
    <xsl:import href = ”takeWhile.xsl” />
    <xsl:import href = ”easyDiffList.xsl” />

    <myImproveGenerator:myImproveGenerator/>
    <myImproveController:myImproveController/>

    <xsl:output indent = ”yes” omit-xml-declaration = ”yes” />

    <xsl:template name = ”improvedDiff” >
        <xsl:param name = ”pFun” select = ”/..” />
        <xsl:param name = ”pX” />
        <xsl:param name = ”pH0” select = ”0.1” />
        <xsl:param name = ”pEpsRough” select = ”0.0001” />
        <xsl:param name = ”pEpsImproved” select = ”0.0000001” />

        <xsl:variable name = ”vMyImproveGenerator”
                      select = ”document('')/*/myImproveGenerator:*[1]” />

        <xsl:variable name = ”vMyImproveController”
                      select= ”document('')/*/myImproveController:*[1]” />

        <xsl:variable name = ”vrtfRoughResults” >
            <xsl:call-template name = ”easyDiffList” >
                <xsl:with-param name = ”pFun” select = ”$pFun” />
                <xsl:with-param name = ”pX” select = ”$pX” />
                <xsl:with-param name = ”pH0” select = ”$pH0” />
                <xsl:with-param name = ”pEps” select= ”$pEpsRough” />
            </xsl:call-template>
        </xsl:variable>

        <xsl:variable name = ”vrtfResults” >
            <xsl:call-template name = ”takeWhile” >
                <xsl:with-param name = ”pGenerator”
                                select = ”$vMyImproveGenerator” />
                <xsl:with-param name = ”pParam0”
                          select = ”msxsl:node-set($vrtfRoughResults)/*” />
                <xsl:with-param name = ”pController”
                                select = ”$vMyImproveController” />
                <xsl:with-param name = ”pContollerParam”
```

```
                              select = ”$pEpsImproved” />
          </xsl:call-template>
      </xsl:variable>

      <xsl:variable name = ”vResults”
                    select = ”msxsl:node-set($vrtfResults)/*” />

      <xsl:value-of select = ”$vResults[last()]/*[2]” />
  </xsl:template>

  <xsl:template name = ”myImproveGenerator”
                match = ”*[namespace-uri()='myImproveGenerator']” >

      <xsl:param name = ”pList” select = ”/..” />
      <xsl:param name = ”pParams” select = ”/..” />

      <xsl:choose>
          <xsl:when test = ”not($pList)” >
              <xsl:call-template name = ”improve” >
                  <xsl:with-param name= ”pList” select=”$pParams” />
              </xsl:call-template>
          </xsl:when>
          <xsl:otherwise>
              <xsl:call-template name = ”improve” >
                  <xsl:with-param name = ”pList”
                                  select = ”$pList[last()]/*” />
              </xsl:call-template>
          </xsl:otherwise>
      </xsl:choose>
  </xsl:template>

  <xsl:template name = ”MyImproveController”
                match = ”*[namespace-uri()='myImproveController']” >
      <xsl:param name = ”pList” select = ”/..” />
      <xsl:param name = ”pParams” select = ”0.01” />

      <xsl:choose>
          <xsl:when test = ”count($pList) < 2” >1</xsl:when>
          <xsl:otherwise>

              <xsl:variable name = ”vDiff”
                            select = ”$pList[last()]/*[2]
                                     - $pList[last() - 1]/*[2]” />

              <xsl:if test = ”not($vDiff < $pParams
                               and $vDiff > (0 - $pParams)
                                )
                              and count($pList) <= 5 “ >1</xsl:if>
          </xsl:otherwise>
      </xsl:choose>
  </xsl:template>
```

```
</xsl:stylesheet>
```

And here's a small example of using `improvedDiff`.

testImprovedDiff.xsl:

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

    <xsl:import href = "improvedDiff.xsl" />

    <myFunction:myFunction/>

    <xsl:output indent = "yes" omit-xml-declaration = "yes" />

    <xsl:template match = "/" >
        <xsl:variable name = "vMyFun"
                      select = "document('')/*/myFunction:*[1]" />
         f = x^2
         x = 15
         h0 = 0.1
         EpsRough = 0.0001
         EpsImproved = 0.000000000001
         f'(x) = <xsl:text/>

        <xsl:call-template name = "improvedDiff" >
            <xsl:with-param name = "pFun" select = "$vMyFun" />
            <xsl:with-param name = "pX" select = "15" />
            <xsl:with-param name = "pH0" select = "0.1" />
            <xsl:with-param name = "pEpsRough" select = "0.0001" />
            <xsl:with-param name = "pEpsImproved"
                            select = "0.000000000001" />
        </xsl:call-template>

          f = x^2
          x = 0.5
          h0 = 0.1
          EpsRough = 0.0001
          EpsImproved = 0.0000000000001
          f'(x) = <xsl:text/>

        <xsl:call-template name = "improvedDiff" >
            <xsl:with-param name = "pFun" select = "$vMyFun" />
            <xsl:with-param name = "pX" select = "0.5" />
            <xsl:with-param name = "pH0" select = "0.1" />
            <xsl:with-param name = "pEpsRough" select = "0.0001" />
            <xsl:with-param name = "pEpsImproved"
                            select = "0.0000000000001" />
        </xsl:call-template>

          f = x^2
          x = 6
          h0 = 0.1
          EpsRough = 0.0001
          EpsImproved = 0.0000000000001
          f'(x) = <xsl:text/>

        <xsl:call-template name = "improvedDiff" >
            <xsl:with-param name = "pFun" select = "$vMyFun" />
            <xsl:with-param name = "pX" select = "6" />
            <xsl:with-param name = "pH0" select = "0.1" />
            <xsl:with-param name = "pEpsRough" select = "0.0001" />
            <xsl:with-param name = "pEpsImproved"
                            select = "0.0000000000001" />
        </xsl:call-template></xsl:template>

    <xsl:template name = "myFunction"
                  match = "*[namespace-uri()='myFunction']" >
        <xsl:param name = "pParam" select = "/.." />
        <xsl:value-of select = "$pParam * $pParam" />
    </xsl:template>

</xsl:stylesheet>
```

Results:

```
f = x^2
x = 15
h0 = 0.1
EpsRough = 0.0001
EpsImproved = 0.000000000001
f'(x) = 30.00000000000057


f = x^2
x = 0.5
h0 = 0.1
EpsRough = 0.0001
EpsImproved = 0.0000000000001
f'(x) = 0.9999999999999998


f = x^2
x = 6
h0 = 0.1
EpsRough = 0.0001
EpsImproved = 0.0000000000001
f'(x) = 12.000000000001286
```

# Numerical integration

In this section we'll provide a solution to the following problem: given a real-valued function **f** which has one real argument, and two endpoints **a** and **b** estimate the area of the shape defined by the graphic of the function, the abscissa axis and the end-points.

The numerical integration algorithm we'd be using can be described in the following steps:

If we think of the curve of the function as a straight line, then the following function computes exactly the area:

```
easyintegrate f a b = (f a + f  b) * (b - a) / 2
```

In practice, the curve will be much different, so the above will provide a satisfactory approximation of the area only on very small intervals. A list of better and better approximations of the area can be obtained by successively dividing (each) interval into two and computing the sum of all subintervals. This is strictly defined by the following Haskell functions:

```
integrate :: Fractional a => (a -> a) -> a -> a -> [a]

integrate f a b = integ f a b (f a) (f b)

integ :: Fractional a => (a -> a) -> a -> a -> a -> a -> [a]

integ f a b fa fb = let (m, fm ) = ((a + b)/2, f m) in

(((fa+fb)*(b-a)/2):
           (map addpair(zip(integ f a m fa fm) (integ f m b fm fb))))
```

where **addpair** is the sum of a pair of numbers, and **zip** produces a list out of two given lists, by making a pair of the n-th elements of each of the two lists the n-th element of the resulting list:

```
zip  ::  [a] -> [b] -> [(a,b)]

zip (x:xs) (y:ys) = (x,y) zip xs ys

zip _ _ = []
```

Because XSLT lacks Haskell's ability of lazy evaluation of infinite lists, we translate the above definition to the following **equivalent algorithm:**

We build a successive list of one dimensional grids, starting with  the interval [a,b], then producing each successive grid from the previous, by including its points and for every two adjacent points $\mathbf{a_k}$, $\mathbf{a_{k+1}}$a new point $\mathbf{a_{mid}} = (\mathbf{a_k} + \mathbf{a_{k+1}}) / 2$. After performing k iterations, the list will be as follows

```
grid a b k  =  [a, a + h, a + 2h, ... a + (2^k -1)*h, b]

where h = (a-b)/2^k
```

Then we define a slight modification of **easyintegrate**, which operates on two consecutive elements of a list:

```
easyintegrate1 f (x:y:ys) = (f x + f  y) * (y - x) / 2
```

Then the **partialSum**, for a particular **grid** will be:

```
partialSum (grid a b k)
                  = foldl (add . (map easyintegrate1)) 0 (grid a b k)
```

The list **partialSums**, for a partial sums over successively obtained **grids** can be now defined:

```
partialSums = [partialSum (grid a b k) | k <- [1..]]
```

In order to eliminate the very inefficient multiple re-calculations of f (once for a point in every iteration),  in practice we will construct another **grid1**:

```
grid1 f a b k
       =  [h, f a, f (a + h), f (a + 2h), ... f (a + (2^k -1)*h), f b]

where h = (a-b)/2^k
```

And we'll use the **easyintegrate2** function

```
easyintegrate2 h (x:y:ys) = (x + y) * h / 2
```

and we change **partialSum** and **partialSums** accordingly, so that **partialSum2** will use easyintegrate2, will pass to it the first element of **grid1** and the tail of **grid1** as arguments.

```
partialSum2 (grid1 f a b k)
              = foldl (add . (map (easyintegrate2 hdgrid)))) 0 tlGrid

where hdgrid = head (grid1 f a b k); tlGrid = tail (grid1 f a b k)

partialSums2 f a b = [partialSum2 (grid1 f a b k) | k <- [1..]]
```

Finally, we can select an appropriate element of **partialSums2** by specifying a tolerance:

```
roughIntegration f a b eps = within eps  (partialSums2 f a b)
```

Here's the corresponding XSLT implementation:

partialSumsList

```
<xsl:stylesheet version = ”1.0”
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:pGenerator="pGenerator"
  xmlns:pController="pController"
  xmlns:IntervalParams="IntervalParams"
  xmlns:mapEasyIntegrate="mapEasyIntegrate"
  xmlns:easy-integrate="easy-integrate"
  exclude-result-prefixes = ”xsl msxsl pGenerator pController
                             IntervalParams easy-integrate” >

   <xsl:import href = ”buildListWhileMap.xsl” />
   <xsl:import href = ”foldl.xsl” />

   <xsl:output indent = ”yes” omit-xml-declaration = ”yes” />

   <pGenerator:pGenerator/>
   <pController:pController/>
   <mapEasyIntegrate:mapEasyIntegrate/>
   <easy-integrate:easy-integrate/>

   <xsl:template name = ”partialSumsList” >
     <xsl:param name = ”pFun” select = ”/..” />
     <xsl:param name = ”pA” />
     <xsl:param name = ”pB” />
     <xsl:param name = ”pEps” select = ”0.001” />

     <xsl:variable name = ”vMyGenerator”
                    select = ”document('')/*/pGenerator:*[1]” />

     <xsl:variable name = ”vMyController”
                   select = ”document('')/*/pController:*[1]” />

     <xsl:variable name = ”vmyEasyIntegrateMap”
                   select = ”document('')/*/mapEasyIntegrate:*[1]” />

     <xsl:variable name = ”vrtfvIntervalParams” >
       <IntervalParams:IntervalParams>
         <Interval>
           <el>
             <xsl:value-of select = ”$pA” />
           </el>
           <el>
             <xsl:value-of select = ”$pB” />
           </el>
         </Interval>
         <xsl:copy-of select = ”$pFun” />
       </IntervalParams:IntervalParams>
     </xsl:variable>

     <xsl:variable name = ”vIntervalParams”
                   select= ”msxsl:node-set($vrtfvIntervalParams)/*” />

     <xsl:variable name = ”vrtfResultIntervalList” >
       <xsl:call-template name = ”buildListWhileMap” >
         <xsl:with-param name = ”pGenerator” select=”$vMyGenerator” />
```

```
        <xsl:with-param name = ″pController″
                        select = ″$vMyController″ />
        <xsl:with-param name = ″pParam0″ select=″$vIntervalParams″ />
        <xsl:with-param name = ″pContollerParam″ select = ″$pEps″ />
        <xsl:with-param name= ″pMap″ select=″$vmyEasyIntegrateMap″ />
      </xsl:call-template>
    </xsl:variable>

    <xsl:copy-of select=″msxsl:node-set($vrtfResultIntervalList)″ />

    <xsl:variable name = ″vResultIntervalList″
      select = ″msxsl:node-set($vrtfResultIntervalList)
                                                    /*[last()]/*″ />

  </xsl:template>

  <xsl:template name = ″listGenerator″
                match = ″*[namespace-uri()='pGenerator']″ >
    <xsl:param name = ″pList″ select = ″/..″ />
    <xsl:param name = ″pParams″ />

    <xsl:variable name= ″pA0″ select=″string($pParams/*[1]/*[1])″ />

    <xsl:variable name= ″pB0″ select=″string($pParams/*[1]/*[2])″ />

    <xsl:variable name = ″pFun″ select = ″$pParams/*[2]″ />

    <xsl:choose>
      <xsl:when test = ″not($pList)″ >
        <xsl:variable name = ″vFa″>
          <xsl:apply-templates select = ″$pFun″ >
            <xsl:with-param name = ″pX″ select = ″$pA0″ />
          </xsl:apply-templates>
        </xsl:variable>

        <xsl:variable name = ″vFb″ >
          <xsl:apply-templates select = ″$pFun″ >
            <xsl:with-param name = ″pX″ select = ″$pB0″ />
          </xsl:apply-templates>
        </xsl:variable>

        <e>
          <xsl:value-of select = ″$pB0 - $pA0″ />
        </e>
        <e>
          <xsl:value-of select = ″$vFa″ />
        </e>
        <e>
          <xsl:value-of select = ″$vFb″ />
        </e>
      </xsl:when>

      <xsl:otherwise>
        <xsl:variable name = ″vprevH″ select=″$pList[last()]/*[1]″ />

        <xsl:variable name = ″vH″ select = ″$vprevH div 2″ />
        <e>
          <xsl:value-of select = ″$vH” />
        </e>

        <xsl:for-each select = ″$pList[last()]/*[position() > 1
                               and position() != last()]” >

          <xsl:variable name = ″vA”
                        select = ″$pA0 + (position() - 1) * $vprevH” />

          <xsl:variable name = ″vMid” select = ″$vA + $vH” />

          <xsl:variable name = ″vF_mid” >
            <xsl:apply-templates select = ″$pFun” >
              <xsl:with-param name = ″pX” select = ″$vMid” />
            </xsl:apply-templates>
          </xsl:variable>

          <xsl:copy-of select = ″.” />

          <e>
            <xsl:value-of select = ″$vF_mid” />
          </e>
        </xsl:for-each>

        <xsl:copy-of select = ″$pList[last()]/*[last()]” />

      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <xsl:template name = ″listController”
                match = ″*[namespace-uri()='pController']” >
    <xsl:param name = ″pList” select = ″/..” />
    <xsl:param name = ″pParams” />

    <xsl:choose>
      <xsl:when test = ″count($pList) < 2” >1</xsl:when>
      <xsl:otherwise>
        <xsl:variable name = ″vLastDiff”
                      select = ″$pList[last()] - $pList[last() - 1]” />

        <xsl:if test = ″not($vLastDiff < $pParams
                           and $vLastDiff > (0 - $pParams))”
          >1</xsl:if>
      </xsl:otherwise>
      </xsl:choose>
  </xsl:template>

  <xsl:template name = ″mapEasyIntegrate”
                match = ″*[namespace-uri()='mapEasyIntegrate']” >
    <xsl:param name = ″pParams” select = ″/..” /><!-- pMapParams -->
```

```
    <xsl:param name = "pDynParams" select = "/.." />
    <!-- NewBaseListElement -->
    <xsl:param name = "pList" select = "/.." />

    <xsl:variable name = "vResult" >
       <xsl:call-template name = "multiIntegrate" >
          <xsl:with-param name = "pList" select = "$pDynParams/*" />
       </xsl:call-template>
    </xsl:variable>

    <xsl:copy-of select = "msxsl:node-set($vResult)" />

  </xsl:template>

  <xsl:template name = "multiIntegrate" >
    <xsl:param name = "pList" select = "/*/*" />

    <xsl:variable name = "vmyeasyIntegrateFn"
                  select = "document('')/*/easy-integrate:*[1]" />

    <xsl:call-template name = "foldl" >
      <xsl:with-param name = "pFunc" select = "$vmyeasyIntegrateFn" />
      <xsl:with-param name = "pList"
                      select = "$pList[position() > 1
                                and position() < last()]" />
      <xsl:with-param name = "pA0" select = "0" />
    </xsl:call-template>

   </xsl:template>

   <xsl:template name = "myEasyIntegrateFn"
                 match = "*[namespace-uri()='easy-integrate']" >
     <xsl:param name = "arg1" select = "0" /><!-- pA0 -->
     <xsl:param name = "arg2" select = "0" /><!-- node -->
     <xsl:value-of select = "$arg1 +
                            (($arg2 + $arg2/following-sibling::*[1])
                               div 2 )
                           * $arg2/../*[1]" />
  </xsl:template>

</xsl:stylesheet>
```

Let's test **partialSumsList** in a concrete example. We want to produce the list of partial sums for:

$$\int_0^1 x^2 \, dx$$

testPartialSumsList.xsl

```
<xsl:stylesheet version = "1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:IntegralFunction="IntegralFunction"
  exclude-result-prefixes = "xsl IntegralFunction" >

    <xsl:import href = "partialSumsList.xsl" />

    <xsl:output indent = "yes" omit-xml-declaration = "yes" />

    <IntegralFunction:IntegralFunction/>

    <xsl:template match = "/" >
        <xsl:variable name = "vMyFun"
                      select = "document('')/*/IntegralFunction:*[1]" />

        <xsl:variable name = "vrtfPartialSums" >
            <xsl:call-template name = "partialSumsList" >
                <xsl:with-param name = "pFun" select = "$vMyFun" />
                <xsl:with-param name = "pA" select = "0" />
                <xsl:with-param name = "pB" select = "1" />
                <xsl:with-param name = "pEps" select = "0.001" />
            </xsl:call-template>
        </xsl:variable>

        <xsl:variable name = "vPartialSums"
                      select = "msxsl:node-set($vrtfPartialSums)/*" />

        <xsl:copy-of select = "$vPartialSums" />
    </xsl:template>

    <xsl:template name = "myIntegralFn"
                  match = "*[namespace-uri()='IntegralFunction']" >
        <xsl:param name = "pX" />

        <xsl:value-of select = "$pX * $pX" />
    </xsl:template>

</xsl:stylesheet>
```

Results of testPartialSumsList.xsl

$$\int_0^1 x^2 \, dx$$

```
0.5 0.375 0.34375 0.3359375 0.333984375
```

As we can see, the last partial sum is an approximation of the true value of the integral (1/3) with the desired accuracy.

A final thought about the above sequence -- it is produced with successive halving of h. Therefore, we can improve it, using the same mechanism as we used in implementing numerical differentiation. The **improvedIntegration** template bellow is almost identical to the **`improvedDiff`** function from section 4.2. We are producing a list of successively improved lists of  partial sums and taking every second element of these lists, until our tolerance criterion has been satisfied.

It is instructive to see how much we are re-using the powerful generic functionality that was already built. Such reuse would simply be impossible without having implemented higher-order functions in XSLT.

improvedIntegration

```
<xsl:stylesheet version = ”1.0”
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:myImproveGenerator="myImproveGenerator"
  xmlns:myImproveController="myImproveController"
  exclude-result-prefixes = ”xsl msxsl myImproveGenerator
                             myImproveController” >

    <xsl:import href = ”improve.xsl” />
    <xsl:import href = ”take.xsl” />
    <xsl:import href = ”takeWhile.xsl” />
    <xsl:import href = ”PartialSumsList.xsl” />

    <myImproveGenerator:myImproveGenerator/>
    <myImproveController:myImproveController/>

    <xsl:output indent = ”yes” omit-xml-declaration = ”yes” />

    <xsl:template name = ”improvedIntegration” >
        <xsl:param name = ”pFun” select = ”/..” />
        <xsl:param name = ”pA” />
        <xsl:param name = ”pB” />
        <xsl:param name = ”pEpsRough” select = ”0.0001” />
        <xsl:param name = ”pEpsImproved” select = ”0.0000001” />

        <xsl:variable name = ”vMyImproveGenerator”
                      select = ”document('')/*/myImproveGenerator:*[1]” />
        <xsl:variable name = "vMyImproveController"
                      select= "document('')/*/myImproveController:*[1]" />

        <xsl:variable name = "vrtfRoughResults" >
            <xsl:call-template name = "partialSumsList" >
                <xsl:with-param name = "pFun" select = "$pFun" />
                <xsl:with-param name = "pA" select = "$pA" />
                <xsl:with-param name = "pB" select = "$pB" />
                <xsl:with-param name = "pEps" select= "$pEpsRough" />
            </xsl:call-template>
        </xsl:variable>

        <xsl:variable name = "vrtfResults" >
            <xsl:call-template name = "takeWhile" >
                <xsl:with-param name = "pGenerator"
                                select = "$vMyImproveGenerator" />
                <xsl:with-param name = "pParam0"
                          select = "msxsl:node-set($vrtfRoughResults)/*" />
                <xsl:with-param name = "pController"
                                select = "$vMyImproveController" />
                <xsl:with-param name = "pContollerParam"
                                select = "$pEpsImproved" />
            </xsl:call-template>
        </xsl:variable>

        <xsl:variable name = "vResults"
                      select = "msxsl:node-set($vrtfResults)/*" />

        <xsl:value-of select = "$vResults[last()]/*[2]" />
    </xsl:template>

    <xsl:template name = "myImproveGenerator"
                  match = "*[namespace-uri()='myImproveGenerator']" >
        <xsl:param name = "pList" select = "/.." />
        <xsl:param name = "pParams" select = "/.." />

        <xsl:choose>
            <xsl:when test = "not($pList)" >
                <xsl:call-template name = "improve" >
                    <xsl:with-param name= "pList" select="$pParams" />
                </xsl:call-template>
            </xsl:when>
            <xsl:otherwise>
                <xsl:call-template name = "improve" >
                    <xsl:with-param name = "pList"
                                    select = "$pList[last()]/*" />
                </xsl:call-template>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>
```

```
    <xsl:template name = "MyImproveController"
                  match = "*[namespace-uri()='myImproveController']" >
        <xsl:param name = "pList" select = "/.." />
        <xsl:param name = "pParams" select = "0.01" />

        <xsl:choose>
           <xsl:when test = "count($pList) < 2" >1</xsl:when>
            <xsl:otherwise>
              <xsl:variable name = "vDiff"
                            select = "$pList[last()]/*[2]
                                   - $pList[last() - 1]/*[2]" />

             <xsl:if test = "not($vDiff < $pParams and $vDiff
                                 > (0 - $pParams))
                              and count($pList) <= 5 " >1</xsl:if>
           </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

</xsl:stylesheet>
```

Let's now test **improvedIntegration** with a concrete example. We want to calculate the following four integrals:

$$\int_0^1 x^2\, dx \ , \quad \int_0^1 x^3\, dx \ , \quad \int_0^1 4/(1+x^2)\ dx\ , \quad \text{and} \quad \int_1^2 1/x\, dx$$

These should return accordingly:

1/3, 1/4, 3.1415926... (Pi), 0.69314718... (ln 2)

testImprovedIntegration.xsl

```
<xsl:stylesheet version = "1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:IntegralFunction="IntegralFunction"
  xmlns:IntegralFunction2="IntegralFunction2"
  xmlns:IntegralFunction3="IntegralFunction3"
  xmlns:IntegralFunction4="IntegralFunction4"
  exclude-result-prefixes = "xsl IntegralFunction IntegralFunction2
                             IntegralFunction3 IntegralFunction4" >

    <xsl:import href = "improvedIntegration.xsl" />

    <xsl:output method = "html" encoding = "UTF-8" />
    <IntegralFunction:IntegralFunction/>
    <IntegralFunction2:IntegralFunction2/>
    <IntegralFunction3:IntegralFunction3/>
    <IntegralFunction4:IntegralFunction4/>

    <xsl:template match = "/" >
        <br/>
 1 <br/>
   <font size = "5" >∫</font>x² =

        <xsl:variable name = "vMyFun"
                      select = "document('')/*/IntegralFunction:*[1]" />

        <xsl:call-template name = "improvedIntegration" >
            <xsl:with-param name = "pFun" select = "$vMyFun" />
            <xsl:with-param name = "pA" select = "0" />
            <xsl:with-param name = "pB" select = "1" />
            <xsl:with-param name = "pEpsRough" select = "0.001" />
            <xsl:with-param name = "pEpsImproved" select = "0.0001" />
        </xsl:call-template>
        <br/>
0

        <br/>
        <br/>&#xA0;
1<br/><font size = "5" >&#x222B;</font>x&#x00B2; =

        <xsl:variable name = "vMyFun2"
                      select = "document('')/*/IntegralFunction2:*[1]" />

        <xsl:call-template name = "improvedIntegration" >
            <xsl:with-param name = "pFun" select = "$vMyFun2" />
            <xsl:with-param name = "pA" select = "0" />
            <xsl:with-param name = "pB" select = "1" />
            <xsl:with-param name = "pEpsRough" select = "0.001" />
            <xsl:with-param name = "pEpsImproved" select = "0.0001" />
        </xsl:call-template>
        <br/>
0

        <br/>
        <br/>&#xA0;
1<br/><font size = "5" >&#x222B;</font> 4/(1 + x&#x00B2;) =

        <xsl:variable name = "vMyFun3"
                      select = "document('')/*/IntegralFunction3:*[1]" />

        <xsl:call-template name = "improvedIntegration" >
            <xsl:with-param name = "pFun" select = "$vMyFun3" />
            <xsl:with-param name = "pA" select = "0" />
            <xsl:with-param name = "pB" select = "1" />
            <xsl:with-param name = "pEpsRough" select = "0.00001" />
            <xsl:with-param name = "pEpsImproved"
```

```
                                        select = "0.0000000000001" />
        </xsl:call-template>
        <br/>
0

        <br/>
        <br/>&#xA0;
2<br/>
        <font size = "5" >&#x222B;</font> 1/x =

        <xsl:variable name = "vMyFun4"
                      select = "document('')/*/IntegralFunction4:*[1]" />

        <xsl:call-template name = "improvedIntegration" >
            <xsl:with-param name = "pFun" select = "$vMyFun4" />
            <xsl:with-param name = "pA" select = "1" />
            <xsl:with-param name = "pB" select = "2" />
            <xsl:with-param name = "pEpsRough" select = "0.00001" />
            <xsl:with-param name = "pEpsImproved"
                                  select = "0.000000001" />
        </xsl:call-template>
        <br/>
1
    </xsl:template>

    <xsl:template name = "myIntegralFn"
                  match = "*[namespace-uri()='IntegralFunction']" >
        <xsl:param name = "pX" />

        <xsl:value-of select = "$pX * $pX" />
    </xsl:template>

    <xsl:template name = "myIntegralFn2"
                  match = "*[namespace-uri()='IntegralFunction2']" >
        <xsl:param name = "pX" />

        <xsl:value-of select = "$pX * $pX * $pX" />
    </xsl:template>

    <xsl:template name = "myIntegralFn3"
                  match = "*[namespace-uri()='IntegralFunction3']" >

        <xsl:param name = "pX" />

        <xsl:value-of select = "4 div (1 + $pX * $pX)" />
    </xsl:template>

    <xsl:template name = "myIntegralFn4"
                  match = "*[namespace-uri()='IntegralFunction4']" >

        <xsl:param name = "pX" />

        <xsl:value-of select = "1 div $pX" />
    </xsl:template>

</xsl:stylesheet>
```

Results of testImprovedIntegration.xsl

$$\int_0^1 x^2 \, dx = 0.3333333333333333$$

$$\int_0^1 x^3 \, dx = 0.25$$

$$\int_0^1 4/(1 + x^2) \, dx = 3.1415926535897096$$

$$\int_1^2 1/x \, dx = 0.693147180943227$$

As can be seen, the accuracy of the calculated Pi is 13 digits after the decimal point, and the accuracy of calculating ln 2 is 9 digits after the decimal point.

## Summary

Not only the purpose of this article to prove that XSLT can be considered a functional programming language has been fulfilled by providing XSLT implementation for the most major FP design patterns and examples from John Hughes article "Why Functional Programming matters" (this article contains the code of 35 functions), but as a side effect we have now available what can be considered the first XSLT functional programming library. The full library code, together with test examples demonstrating the use of individual functions, is available at the downloads page of TopXML.COM as specified in the links at the start of this article. The fxsl library and its next versions are (will be) available also as part of the evaluation version of the Xselerator IDE, which can be downloaded at http://www.topxml.com/xselerator/ . This is done with the purpose to assure that the new **fxsl** library releases would be made constantly available to the thousands of highly experienced XSLT programmers, who are users of Xselerator. Anyone, who is interested to study and start using the **fxsl** library, can do this now.

### Discussion

We have provided a multitude of examples illustrating the usefulness of higher-order functions in XSLT. Most of the examples from John Hughes article "Why Functional Programming matters" have been implemented and the XSLT code demonstrated. Many of them (e.g. foldl, foldr, foldl-tree, map, buildListWhile, buildListWhileMap) are important functional programming design patterns that can generate unlimited types of new functions, depending on the functions passed to them as parameters.

All this is more than an ample proof that XSLT **is** in fact a full-pledged functional programming language.

On the other side, the XSLT code of those functions seems too-verbose compared to the corresponding Haskell code. The process of writing functional XSLT code can be made much more straightforward and easier by providing support for higher-order functions in XPath and XSLT, thus definitely improving even further the compactness, readability and reliability of XSLT functional code. It is the ideal time right now for the W3C XPath 2.0 working group to make the decision to provide the necessary support for higher-order functions as part of the standard XPath 2.0 specification. In case this golden opportunity is missed, then generic templates and libraries will be used in the years to come.

### Resources

**[1]Why Functional Programming Matters**

By John Hughes

A brilliant paper, explaining the essence and beauty of the most important FP concepts. Multiple, very nice examples. Beautiful and elegant -- this is a "must read" for everyone who needs to understand what FP is all about.

[2] Functional Programming and XML

**By Bijan Parsia:** Describes support for XML in some FP languages: HaXML, LambdaXML and Erlang.

**[3] What kind of language is XSLT?**

**By Michael Kay:**("Note: Although XSLT is based on functional programming ideas, it is not as yet a full functional programming language, as it lacks the ability to treat functions as a first-class data type.")

[4] A formal semantics of patterns in XSLT (PDF)

By Philip Wadler

[5] Philip Wadler's home page

A researcher with major contributions in the area of formal definition of the semantics of XSLT and XPath

[6] Haskell home page

The home page of the FP language Haskell.

[7] Haskell, The Craft of Functional Programming, Second Edition

By Simon Thompson

**[8]** XSL Transformations (XSLT) Version 1.0W3C Recommendation 16 November 1999

[9] Personal communication with Corey Haines, through messages in the xsltalk forum.

[10] A Gentle Introduction to Haskell, Version 98

By Paul Hudak, John Peterson, Joseph Fasel

## About the author

Dimitre Novatchev is the author of the popular tool for learning XPath -- the XPath Visualizer, and he developed the first systematic implementation of the concept of "*generic templates*". His background (and PhD, a long time ago) is in computer science and mathematics. He has been working at Fujitsu Australia on various aspects of OODBMS, including XML support for Jasmine. At the time of writing Dimitre is a software development manager at the UN Centre in Vienna, Austria.