XMLMath 1.1 Manual

Erik van Zijst
erik@prutser.cx

June 2006

Contents

1 Introduction

This document contains the manual of xmlmath. It consists of three logical parts. Chapter 2 shows what a basic xmlmath expression looks like and how it is fed to xmlmath's console-based expression evaluator to calculate the result.

Chapters 4 to 9 then go on to describe the various parts of xmlmath expressions. They cover declarations, stanzas and inclusion of external documents. Various built-in operators are also discussed. The latter is used to compose xml-based libraries of frequently used equations and formulas.

Finally, chapter 10 describes how xmlmath is used as a library inside other programs.

2 Basic Expression Structure

An xmlmath expression consists of a tree of nested xml tags. Each tag reads the value of its child nodes, applies an operation to it and returns the result to its parent tag. This is illustrated below with the expression that calculates 1 + 1.

```
<expression xmlns="http://xmlmath.org/1.0">
  <add>
    <long value="1"/>
    <long value="1"/>
  </add>
</expression>
```

Note: *Because the xmlmath expression parser validates the input files against the xmlschema, it is necessary to always include the proper xml namespace directive in the expression's root tag.*

2.1 System Requirements

Xmlmath is written in Java 1.5 and as such requires a 1.5 runtime environment. Because xmlmath makes use of specific 1.5 features, older Java environments are currently not supported.

Although xmlmath uses the open-source xmlbeans library for parsing its expression files, it is not necessary to install xmlbeans separately: the binary distributions of xmlmath have a built-in copy of xmlbeans.

2.2 Running XMLMath Standalone

The xmlmath package comes with a standalone evaluation engine that is used to run expressions from the shell or command line.

To evaluate the expression from the previous paragraph, save it in a file and run it through the evaluator. Because xmlmath reads from standard input, you have to use a pipe. On unix or linux, type the following:

```
$ cat file.xml | java -jar dist/xmlmath-VERSION.jar
```

or:

```
$ java -jar dist/xmlmath-VERSION.jar < file.xml
```

When xmlmath is used in scripts or manually from the command line, it may be convenient to first start the evaluator and then type the expression, followed by a termination string to close the input:

```
$ java -jar xmlmath-1.1-SNAPSHOT.jar << EOF
> <expression xmlns="http://xmlmath.org/1.0">
>   <add>
>     <long value="1"/>
>     <long value="1"/>
>   </add>
> </expression>
> EOF
1.0
$
```

The latter is sometimes used in this manual.

3 Datatypes

Each tag (or operator) returns an instance of a particular datatype. By default, xmlmath understands the following, hierarchical types:

- value
  - boolean
  - number
    - long
    - double
  - string
  - list

The operator's datatype defines whether or not it can understand the value returned by another operator. For example, the <add> operator from the previous example only understands numerical child elements of type number. Because datatypes are hierarchical, the <add> operator also understands child elements of type double and long. An operator's return value can differ from the type of its child elements (also known as arguments or operands). The remainder of this section describes the various datatypes.

3.1 Boolean

Boolean logic is supported by xmlmath through a range of boolean operators including: <and>, <or>, <not>, <xor>, <true> and <false>. The example below shows a boolean expression that evaluates to true.

```
<expression xmlns="http://xmlmath.org/1.0">
  <and>
    <true/>

    <not>
      <equals>
        <false/>
        <true/>
      </equals>
    </not>

    <xor>
      <false/>
      <true/>
    </xor>
  </and>
</expression>
```

3.2 String

The string type simply represents one or more lines of text. Using string operators, these can be manipulated. The example below creates a new string by concatenating a number of smaller strings:

```
<expression xmlns="http://xmlmath.org/1.0">
  <strcat>
    <string value="Hello"/>
    <string value=" "/>
    <string value="world."/>
  </strcat>
</expression>
```

Below is another example that uses <substr> and <strlen> to transform the string "unhappy" into "happy". The operator <substr> takes three arguments: first a string and then two numerical (long) indices. The first marks the beginning index (inclusive), while the second marks the ending index (exclusive). Note that it is also legal to omit the second index. When omitted, the operator lets the selected substring extend to the end of the input string.

```
<expression xmlns="http://xmlmath.org/1.0">
  <substr>
    <string value="unhappy"/>
    <long value="2"/>
    <strlen>
      <string value="unhappy"/>
    </strlen>
  </substr>
</expression>
```

Note: The <toString> operator is used to turn any value into a string.

3.3 Long

The long type represents a 64-bit, signed integer number, using two's complement arithmetic. A long in xmlmath behaves identical to a long in Java.

3.4 Double

The double type represents a 64-bit, IEEE 754 signed floating point number. A double in xmlmath behaves identical to a double in Java.

Many xmlmath operators work with type number, rather than long or double. This allows them to handle any numerical value. Almost all these operators default to floating point arithmetic, even when integer values are provided as arguments. The reason for this is that an operator does not dynamically inspect the data type of its arguments. In some cases it is necessary to retain the exact precision of integer numbers to avoid rounding errors. Use the attribute "datatype" to explicitly instruct an operator to use either floating point, or integer arithmetic. The example below shows the effect on a simple division.

```
$ java -jar xmlmath-1.1-SNAPSHOT.jar << EOF
> <expression xmlns="http://xmlmath.org/1.0">
>   <quotient>
>     <long value="4"/>
>     <long value="2"/>
>   </quotient>
> </expression>
> EOF
2.0

$ java -jar xmlmath-1.1-SNAPSHOT.jar << EOF
> <expression xmlns="http://xmlmath.org/1.0">
>   <quotient datatype="long">
>     <long value="4"/>
>     <long value="2"/>
>   </quotient>
> </expression>
> EOF
2
```

Regardless of the type of the arguments to <quotient>, when the datatype is explicitly set to "long", the result of the computation is an exact number. When using integer arithmetic it is important to realize that floating point arguments are first truncated (not rounded) to integer values.

3.5 List

The list datatype is used to hold an array of items. These lists can be manipulated with the list operators. Lists can contain any number of elements and are not necessarily homogeneous, meaning that a list can contain elements of different types. Lists maintain the original order of the elements. By default, xmlmath comes with a number of list operators, including: <sort>, <unique>, <for>, <listSum> and <listProduct>. The latter two of these compute the combined sum and product of all list items respectively. To create a list of elements, use the <list> operator. This is illustrated below:

```
<expression xmlns="http://xmlmath.org/1.0">
   <list>
    <long value="1"/>
    <string value="foo"/>
    <double value="2.55"/>
    <true/>
    <pi/>
    <e/>
    <infinity/>
  </list>
</expression>
```

When this expression is evaluated, xmlmath prints the individual elements as a comma-separated, ordered list:

```
$ java -jar xmlmath-1.1-SNAPSHOT.jar < samples/list.xml
[1, foo, 2.55, true, 3.141592653589793, 2.718281828459045, Infinity]
```

List items can be accessed individually using the operator <listItem> which takes both a list and an index number and returns the element at the specified position in the given. This can be useful in combination with a loop to apply an operation to each item of a list. This will be further discussed in section "XMLMath Core Operators".

The operator <sort> is used to sort the elements in a list ascending or descending. However, sorting is only possible if all elements are of the same type. Trying to sort a heterogeneous list will result in an `EvaluationException` being thrown by xmlmath:

```
$ java -jar xmlmath-1.1-SNAPSHOT.jar << EOF
> <expression xmlns="http://xmlmath.org/1.0">
>   <sort>
>     <list>
>       <long value="1"/>
>       <string value="foo"/>
>       <double value="2.55"/>
>     </list>
>   </sort>
> </expression>
> EOF
Cannot sort a heterogeneous list. Make sure all elements are of the same
type.
```

By default the xmlmath evaluator suppresses the detailed stacktrace of the exception. Use the `-e` flag to let xmlmath dump the complete exception to stderr. This may sometimes help to locate the problem in a long expression.

```
$ java -jar xmlmath-1.1-SNAPSHOT.jar -e << EOF
> <expression xmlns="http://xmlmath.org/1.0">
>   <sort>
>     <list>
>       <long value="1"/>
>       <string value="foo"/>
>       <double value="2.55"/>
>     </list>
>   </sort>
> </expression>
> EOF
org.xmlmath.operands.EvaluationException: Cannot sort a heterogeneous list.
Make sure all elements are of the same type.
        at org.xmlmath.operands.Sort.getList(Sort.java:46)
        at org.xmlmath.operands.AbstractListValue.getValue(AbstractListValue
.java:10)
        at org.xmlmath.operands.ExpressionImpl.evaluate(ExpressionImpl.java:
55)
        at org.xmlmath.Evaluator.main(Evaluator.java:67)
```

3.6 Value

The value-type is the base type of all datatypes. Operators whose arguments are of type "value" can therefore accept all types. An example of an operator that works with type "value" is <equals>, which can compare any two elements and returns true if they are of the same type and represent the same value. This means that `<long value="2"/>` can be compared to `<double value="2"/>`, but will evaluate to false.

3.7 Typecasting

Xmlmath's hierarchical type system is based on inheritance, where subtypes inherit the properties of their parent. This makes it possible to pass an instance of a subtype to an operator. An example of this is how many numerical operators such as <subtract> take child elements of type number, which is the base-type for long and double. Similarly, some operators, including <equals>, or <isNil> accept the base-type of all datatypes: value.

In some cases one might end up in a situation where the datatype of an operator is less specific than what is required by its parent operator. Consider the example below. It attempts to computes the sine of 2pi, but fails with a (somewhat cryptical) message that the expression is semantically incorrect:

```
$ java -jar xmlmath-1.1-SNAPSHOT.jar -e << EOF
> <expression xmlns="http://xmlmath.org/1.0">
>   <sin>
>     <product datatype="double">
>       <double value="2"/>
>       <pi/>
>     </product>
>   </sin>
> </expression>
> EOF
Expression violates the schema: [error: cvc-complex-type.2.4a: Expected elem
ent 'abstract-double@http://xmlmath.org/1.0' instead of 'product@http://xmlm
ath.org/1.0' here in element sin@http://xmlmath.org/1.0, error: cvc-complex-
type.2.4c: Expected element 'abstract-double@http://xmlmath.org/1.0' before
the end of the content in element sin@http://xmlmath.org/1.0]
$
```

The reason for this error is that the <sin> operator requires a child element of type double, while <product> is of type number. However, because we know that <product> returns a double, we can use explicit typecasting to indentify the result of <product> as a double:

```
$ java -jar dist/xmlmath-1.1-SNAPSHOT.jar << EOF
> <expression xmlns="http://xmlmath.org/1.0">
>   <sin>
>     <castDouble>
>       <product datatype="double">
>         <double value="2"/>
>         <pi/>
>       </product>
>     </castDouble>
>   </sin>
> </expression>
> EOF
-2.4492935982947064E-16
$
```

Note: Remember that floating point arithmetic is not precise, which is why the result is not exactly 0, but rather a value very close to 0.

Note that typecasting does not convert the data to a different representation. Instead, it merely offers a way to tell the parser and evaluation engine that a value is really of a different type. Typecasting is supported using the operators <castLong>, <castDouble>, <castString>, <castBoolean>, <castList> and <castNumber>. Trying to cast a value to an incompatible type will result in a `TypeCastException` being thrown.

Xmlmath also offers some rudimentary support for converting one value to another datatype. This was shown previously using the <toString> operator, which returns a string representation of any value.

4 Declarations & Scopes

Xmlmath allows values to be labeled with a name, so they can be referred to further down the expression tree. This is a bit like a variable in a programming language. The example below declares a variable x and then computes x + x + x:

```
<expression xmlns="http://xmlmath.org/1.0">
  <add>

    <declare name="x">
      <long value="5"/>
    </declare>

    <linkLong name="x"/>
    <linkLong name="x"/>
    <linkLong name="x"/>
  </add>
</expression>
```

The <declare> operator's counterpart is <link>. The first is used to define a variable, the latter is used to refer to its value from a different, deeper location in the expression. Note that there are separate link tags for each datatype. For example, when the declaration's body returns a double, <linkDouble> can be used to refer to it. When a declaration's datatype is unknown or unimportant, <link> can be used. When a link tag is used to refer to an incompatible declaration, a `TypeCastException` is thrown.

Variables can be declared in any operator, as long as they precede the operator's arguments. When a variable is declared, it is accessible to all operators further down the expression tree, as well as to the operator in which the declaration was made (which was the case in the above example).

In contrast to most variables in programming languages, declared values in xmlmath cannot be assigned a value other than the value represented by the declaration's child element. Xmlmath does not have an assignment operator.

Declarations at different levels in the expression tree can have the same name. In such case, a <link> operator always refers to the most recently declared instance. This is illustrated in the example below which prints "bar" instead of "foo":

```
<expression xmlns="http://xmlmath.org/1.0">
  <declare name="x">
    <string value="foo"/>
  </declare>
  <toString>
    <declare name="x">
      <string value="bar"/>
    </declare>
    <linkString name="x"/>
  </toString>
</expression>
```

It is not possible to make declarations with the same name at the same level. Trying to do so will raise a `ParseException`.

The visibility of a declaration is defined by its scope. Each time the expression parser parses a new operator, it creates a new scope. As the parser traverses the expression tree, these scopes form a stack.

Declarations are attached to the scope of the tag in which they are created. When a <link> tag further down the expression refers to a declaration, it is resolved through that tag's local scope. If the declaration is not found in that scope, the search continues in the parent scope. This hierarchical search continues until either the declaration is found in a scope deeper in the stack, or the root scope is reached. In the latter case a `ParseException` is thrown and the following message is printed to stderr: `Reference attempted to undeclared variable x`.

5 Stanza's

Declarations provide a valuable feature to xmlmath expressions, as they allow the result of sub-expression to be labeled with a name and then re-used at one or more other locations in the expression. However, xmlmath comes with another important declarative operator: <stanza>.

A stanza also offers a way to describe and label a sub-expression, but unlike declarations, stanzas are not evaluated in the scope where they are created. Instead, stanzas are used through the <inline> tag, which is substituted for the stanza's body during the parsing phase. By the time the parsing phase is completed and evaluation begins, all <stanza> and <inline> tags have been removed. Each <inline> tag has then been replaced by a copy of its stanza.

The effect of inlining a stanza is illustrated below. The illustration shows an expression prior to the parsing phase and the same expression after parsing.

```
<expression xmlns="http://xmlmath.org/1.0">
  <stanza name="foo">
    <string value="bar"/>
  </stanza>
  <inline name="foo"/>
</expression>


<expression xmlns="http://xmlmath.org/1.0">
  <string value="bar"/>
</expression>
```

Stanza's and declarations are independent entities and can therefore use the same names unambiguously.

Because stanzas are never evaluated in the scope where they were created, it is possible to refer to declarations and other stanzas that may not yet be declared. This is illustrated below in an expression that contains a stanza which refers to an integer value "x" to calculate its squared value. Variable "x" is only declared further down the expression tree, just before the stanza is inlined:

```
<expression xmlns="http://xmlmath.org/1.0">

  <stanza name="squared">
    <product datatype="long">
      <linkLong name="x"/>
      <linkLong name="x"/>
    </product>
  </stanza>

  <strcat>
    <string value="4 squared = "/>
    <toString>
      <declare name="x">
        <long value="4"/>
      </declare>
      <inline name="squared"/>
        </toString>
      </strcat>
</expression>
```

As with declarations, stanzas are resolved through the parser scope and when multiple stanzas exist with the same name, the most recently parsed stanza is inlined.

It is possible to refer to a stanza from inside another stanza. However, because a stanza's body substitutes all <inline> tags that refer to it, recursion is not possible. Attempting to let a stanza's body inline itself will cause the parser to crash with a `StackOverflowError`. Circular references between stanzas is unsupported for the same reason.

Similar to declarations, new stanzas can be defined inside each operator tag, including the <stanza> and <declare> tag. The example below shows how this can be used to create inner-stanzas (stanzas only known and accessible inside another stanza). When the example is evaluated, it declares a double named "x", then inlines the stanza "foo", which declares a double "y" which links to "x". Then the inner-stanza "inner_foo" is inlined which returns the value of "y":

```
<expression xmlns="http://xmlmath.org/1.0">

  <stanza name="foo">
    <stanza name="inner_foo">
      <linkDouble name="y"/>
    </stanza>
    <declare name="y">
      <linkDouble name="x"/>
    </declare>
    <inline name="inner_foo"/>
  </stanza>

  <declare name="x">
    <double value="2"/>
  </declare>

  <inline name="foo"/>
</expression>
```

Similar to declarations, stanzas are not typed, meaning that the parser cannot see what datatype it will return at runtime. Therefore, the user has to select the proper <inline> tag if the type is important. The <inline> tag for example returns datatype `value` which is the base type of all datatypes. However, this is not suitable when inline's parent tag requires a more specific datatype. In that case, the user has to use one of inline's typed alternatives such as <inlineLong> or <inlineList>. The datatype returned by the stanza has to be compatible with the type of the inline tag that is used, or a `TypeCastException` is thrown.

6 Including External Documents

It is often useful to define certain functionality in a stanza and then use that stanza in different expressions, without having to duplicate the stanza in each expression. This is possible by putting these stanzas and declarations in a so-called *includes-file* and then using the <include> tag to import that file's contents in an expression.

An includes-file differs from an expression file in its root tag. The root tag of an expression that can be evaluated is always <expression>, while the root tag of an includes-file is always <includes>. Includes-files can only be referred to from expression files or other includes-files, and cannot be evaluated separately. This is because they can only contain the declarative elements <declare> and <stanza>, as well as references to other includes-files. In this respect, includes-files can be compared to libraries in programming languages: they can only be used by real programs or other libraries, but cannot be run individually.

The example below contains a typical includes-file. It declares a number of constants that have wide application, as well as a stanza that performs a generic computation.

```
<includes xmlns="http://xmlmath.org/1.0">
  <stanza name="squared">
    <product datatype="long">
      <linkLong name="x"/>
      <linkLong name="x"/>
    </product>
  </stanza>
  <declare name="avogadro">
    <double value="6.0221415E23"/>
  </declare>
  <declare name="lightspeed">
    <long value="299792458"/>
  </declare>
</includes>
```

The order of the elements is important. Stanzas always precede the declarations. The example below shows how the includes-file is used in an expression that uses one of the file's declared constant values. Given the approximate value of 40000km as the earth's diameter, it calculates how fast a ray of light can travel the earth's entire equator.

```
<expression xmlns="http://xmlmath.org/1.0">

  <include>path/to/includes-file.xml</include>

  <strcat>
    <string value="In a vacuum, it takes light "/>
    <toString>
      <quotient>
        <double value="40000E3"/>
        <linkLong name="lightspeed"/>
      </quotient>
    </toString>
    <string value=" seconds to travel the diameter of the earth."/>
  </strcat>
</expression>
```

Including a file with <include> requires a URL as the body of the <include> tag. This URL must be compatible with the syntax described by RFC's 2396 and 2732. When the protocol is omitted from the URL (as is the case in the above example), the value is assumed to represent a file on the local filesystem.

When relative URLs are used to refer to other files, they are relative to the absolute location of the current file. For example if an includes file is hosted on a webserver, one would use "<include>http://acme.com/includes.xml<include>" to include the file in an expression. However, when the includes file on the webserver refers to another includes file on the same webserver, it can omit the protocol and hostname and use the relative url "<include>includes2.xml<include>".

7 Summary of Declarative Elements

In xmlmath all tags represent a value and by nesting them an expression is created that can be evaluated. This rule applies to all tags, except the three declarative tags: *code*<include>*/code*, *code*<stanza>*/code* and *code*<declare>*/code*. These tags are not part of the evaluation path, unless they are explicitly referred to from another place in the expression.

The three declarative tags form an ordered element group that is supported by every tag, including the declarative tags themselves. They must always precede all other child tags.

8 XMLMath Core Operators

8.1 Conditional Evaluation

8.2 Loops

8.3 Sum

8.4 Input Parameters