DRB 2.2 RC-12

Users Manual

2.2 25/11/2005

Part

1

Chapter 1: Introduction

Because large amounts of information are handled by the "Data Request Broker" (DRB), the ability to intelligently query the supported data sources becomes increasingly important. One of the great strengths of DRB is its flexibility in representing many different kinds of information from diverse sources. To exploit this flexibility, a component of DRB package implements a query engine that benefits from the emerging XQuery Language to provide features for retrieving and interpreting information from these data in a uniform way.

Example

```
<html>
   <body>
       <table>
       {
          for $row in doc("http://server/directory/archive.jar")/*
          return
             <tr>
                <td>{name($row)}</td>
                <td>{$row/@size}</td>
             </tr>
       }
       </table>
   </body>
</html>
```

The previous example illustrates how consise and effective an XQuery combined with DRB for producing an HTML page listing in a table, the name and size of a Jar archive entries. This example shows a very short part of the capabilities of the DRB XQuery engine.

XQuery Language

XQuery is a query language initially designed for querying XML documents. Detached from XML specificities, it provides however a large amount of features for any data source broken down in a tree of nodes and leaves. Almost all data sources have such hierarchical structure and thanks to DRB, it is possible to benefit from XQuery properties applied to these sources.

XQuery is designed to meet the requirements identified by the W3C XML Query Working Group. It is designed to be a language in which queries are concise and easily understood. It is also flexible enough to query a broad spectrum of information sources, including both databases and documents. The Query Working Group has identified a requirement for both a human-readable query syntax and an XML-based query syntax. XQuery is designed to meet the first of these

requirements. XQuery is derived from an XML query language called "Quilt", which in turn borrowed features from several other languages, including "XPath 1.0", "XQL", "XML-QL", "SQL" and "OQL [ODMG]".

XQuery is currently under development by the World Wild Web Consortium (W3C) Members. The language specifications have however, reached a mature level of consolidation and it is possible today to consider that the language basics are stable enough to be implemented. The DRB implementation of XQuery is based on the "XQuery 1.0: An XML Query Language - W3C Working Draft 15 September 2005".

See also: The XQuery specifications can be downloaded from http://www.w3.org/TR/2005/WD-xquery-20050915/ and more generally the latest document can be downloaded from http://www.w3.org/TR/xquery/

Compliance

The DRB implementation of XQuery intends to be compatible with the specifications of the W3C. This means all the features provided by XQuery are not already implemented, but the departures from the specifications have been reduced to the minimum. As a summary, the implementation is a sub-set of the XQuery specifications.

The next chapters describe the supported features of current version of the XQuery engine. Notes and cautions annotate them when the implementation does not comply with or derives from the XQuery specifications.

Grammar notation

In order to remain as concise as possible, the next chapters provide the grammar rules as BNF productions. The definition of the BNF language is outside the scope of this document. If you are not familiarized with BNF, its basics can be summarized as follow:

The grammar rules are broken down in to productions that interconnect to define the language. A set of alphabetical characters expresses the productions. The production definitions are assigned with the "::=" symbol.

Example: A production

```
LeftBrace ::= "{"
```

The previous example shows a production named "LeftBrace" to which is assigned the left brace "{" character. This means, when the language enables "LeftBrace", the text shall contain a left brace.

Logical symbols may link the productions to set-up their relationship.

Example: Interconnected production

AnyBrace ::= "{" | RightBrace RightBrace ::= "}"

In this instance, the production "AnyBrace" matches a left brace as in the previous example or ("|" symbol) matches the "RightBrace" which itself matches a right brace. If the "{" and "RightBrace" were not separated by a "|" but only by spaces, "AnyBrace" would have matched the strings "{ }".

It is possible to restrict the enabled occurrence of productions assigned with the characters "+", "*", "[" and "]" or "?". Where "+" authorizes from one to an unlimited number of occurrences, "*" authorizes from 0 to an unlimited number of occurrences, "[" and "]" (embraced production) means the production is optional. The "?" is formally equivalent to the braces.

Occurrences of assigned production

```
Expression ::= Embraced ( "," Embraced )*
Embraced   ::= LeftBrace (Content | Embraced) RightBrace
Content    ::= [a-Z]+
LeftBrace  ::= "{"
RightBrace ::= "}"
```

The previous example defines the language as an "Expression" that is composed of one "Embraced" production possibly followed by an unlimited number of production composed of a "comma" followed by an "Embraced" production. The "Embraced" production is easy to understand. It also illustrates that use of recursive definition. The "Embraced" is actually defined as a "Content" or another "Embraced" productions preceded and followed by braces. The "Content" definition is defined as any character from "a" to "Z" (ordered in the ASCII table) that can be indefinitely repeated. Finally the defined language may match the following text.

Example: Matching text

{ acontent }, { { embracedcontent } }

You should now understand that the BNF language is very efficient to define the grammatical rules of a language. BNF is used among the next chapters to provide a formal and normative definition even if they are always accompanied by narrative explanations.

Applications

DRB has been initially designed for Space Engineering and Earth Observation applications. A set of examples and scenarios are provided in the chapter "Example Applications". It is a good idea to have a quick overview of these examples to understand the interest of XQuery if you are not familiar with query concepts.

Chapter 2: XQuery Basics

The basic building block of XQuery is the expression. The language provides several kinds of expressions which may be constructed from keywords, symbols, and operands. In general, the operands of an expression are other expressions. XQuery is a functional language which allows various kinds of expressions to be nested with full generality. It is also a strongly-typed language in which the operands of various expressions, operators, and functions must conform to designated types.

Like XML, XQuery is a case-sensitive language. All keywords in XQuery use lower-case characters.

Expression

```
Expr ::=  LogicalExpr
       |  GeneralComp
       |  ArithmeticExpr
       |  UnaryExpr
       |  PrimaryExpr
       |  RangeExpr
       |  UnionExpr
       |  IntersectExceptExpr
       |  ElementConstructor
       |  FLWORExpr
       |  IfExpr
       |  PathExpr
```

An XQuery is always bind to an expression that shall match the "Expr" production defined above. The value of an expression is always a sequence. A sequence is an ordered collection of zero or more items. An item is either a simple value or a node. A simple value consists of a value contained in the value space of one of the primitive data types described in the part named "Data definition with XML-Schema". A node is an information extracted from the data source or constructed in the query. Nodes have typed values (if any), names and attributes, which can be extracted from the node using accessors. The typed value of a node is a sequence of zero or more simple values.

A sequence containing exactly one item is called a singleton sequence. A sequence containing zero items is called an empty sequence.

The productions composing the "Expr" are completely defined in the next chapters.

Note: The current implementation does not support the productions SortExpr, QuantifiedExpr, TypeswitchExpr, ValueComp, NodeComp, OrderComp,

IntersectExceptExpr and CastExpr defined in the XQuery specifications.

Evaluation Context

The evaluation context of an expression is defined as information that is available at the time the expression is evaluated. The assessment context consists of the components listed below.

The first three components of the evaluation context (context item, context position and context size) are called the focus of the expression. The focus enables the processor to keep track of which nodes are being processed by the expression.

The focus for the outermost expression is supplied by the environment in which the expression is evaluated. Certain language constructs, notably the path expression E1/E2,create a new focus for the evaluation of a sub-expression. In these constructs, E2 is evaluated once for each item in the sequence that results from evaluating E1. Each time E2 is evaluated, it is evaluated with a different focus. The focus for evaluating E2 is referred to below as the inner focus, while the focus for evaluating E1 is referred to as the outer focus. The inner focus exists only while E2 is being evaluated. When this evaluation is complete, evaluation of the containing expression continues with its original focus unchanged.

Context Item: The "context item" is the item currently being processed. An item is either a simple value or a node. When the context item is a node, it can also be referred to as the "context node". The context item is returned by the expression ".". When the path expression E1/E2 is evaluated, each item in the sequence obtained by evaluating E1 becomes the context item in the inner focus for an evaluation of E2.

Context Position: The "context position" is the position of the context item within the sequence of items currently being processed. It changes whenever the context item changes. Its value is always an integer greater than zero. The context position is returned by the expression position(). When the expression E1/E2 is evaluated, the context position in the inner focus for an evaluation of E2 is the position of the context item in the sequence obtained by evaluating E1. The position of the first item in a sequence is always 1 (one). The context position is always less than or equal to the context size.

Context Size: The "context size" is the number of items in the sequence of items currently being processed. Its value is always an integer greater than zero. The context size is returned by the expression last(). When the expression E1/E2 is evaluated, the context position in the inner focus for an evaluation of E2 is the number of items in the sequence obtained by evaluating E1.

- **Dynamic Variables:** This is a set of named and typed values. The value of a variable is, in general, a sequence. The types and values of variables are provided by the execution of the XQuery expressions in which the variables are bound.

Type Conversions

Certain operators, functions, and syntactic constructs expect a value of a particular type to be supplied: this is referred to as a required type. A required type may be specified in the following ways:

- In the signature of a function definition
- In the description of an operator
- In the description of a syntactic construct, such as the test in a conditional expression

The required type may be specified as a simple one, an optional occurrence of a simple type, a sequence of a simple type, or a sequence of nodes. If no required type is specified, the value may be any sequence. If the value supplied is not conformed to the required type, it is converted to the required type by applying the basic conversion rules defined below.

In some cases, the basic conversion rules may invoke a type exception (i.e. an error). A type exception may be invoked either statically (during query parsing and analysis) or dynamically (during query execution). As specified by XQuery, the DRB query engine treats a type exception as an unrecoverable error.

Basic Conversion Rules

The basic conversion rules are as follow:

- If the required type is a (i.e. one) simple type or an optional occurrence of a simple type:
  - If the given value is a sequence of more than one item, a type exception is invoked. If the given value is a single node, its typed value is extracted by calling its typed value accessor. If the node has no typed value accessor or if its typed value is a sequence containing more than one item, a type exception is invoked.
  - If the (given or extracted) value does not conform to the required type, a type exception is raised. This includes the case in which the (given or extracted) value is an empty sequence and the required type is not an optional occurrence.

> **Note:** The case of "unknown simple type" described in the XQuery specifications never occurs in DRB because all values are strongly typed.

If the required type is a sequence of a simple type:

If the given value is a sequence containing one or more nodes, each node is replaced by its typed value, resulting in a sequence of simple values. Each of these simple values then is converted to the required type by applying the rules described above for converting values into a required simple type.

If the required type is a sequence of nodes:

If the given value contains any item that is not a node, a type exception is invoked.

Chapter 3: Modules and prologs

A module is a fragment of XQuery code that can independently undergo the analysis phase. A module that contains a Prolog followed by a Query Body is called a main module. A query has exactly one main module. In a main module, the Query Body can be evaluated, and its value is the result of the query. A module that contains a module declaration followed by a Prolog is called a library module. A library module cannot be evaluated directly; instead, it provides function and variable declarations that can be imported into other modules. No module may contain both a module declaration and a Query Body.

A Prolog is a series of declarations and imports that create the environment for query processing. Each declaration or import is followed by a semicolon. A Prolog may include declarations of namespaces, variables, functions, and various processing options. Declarations and imports may be specified in any order, except that variable declarations must avoid circular definitions

Prolog

```
(: An example of an XQuery Prolog :)

xquery version "1.0";

declare xmlspace strip;

declare namespace gael = "http://www.gael.fr";
declare namespace esa = "http://www.esa.int";

declare variable $x as xs:integer external;
```

The Query Body, if present, consists of an expression that defines the result of the query. Evaluation of expressions is described in the next chapter "Expressions". A module can be evaluated only if it has a Query Body.

Version declaration

Any module may contain a version declaration. If present, the version declaration occurs at the beginning of the module, and identifies the applicable XQuery syntax and semantics for a module. The version number "1.0" indicates the requirement that the query must be processed by an XQuery Version 1.0 processor. If the version declaration is not present, the version is presumed to be "1.0". The XQuery engine of DRB raises a static error when processing a query labeled with a version different from "1.0". It is the intent of the XQuery working group to give later versions of this specification numbers other than "1.0", but this intent does not indicate a commitment to produce any future versions of XQuery, nor if any are produced, to use any particular numbering scheme.

```
VersionDecl ::= "xquery" "version" StringLiteral Separator
```

Example: Version declaration

xquery version "1.0";

Module declaration

A module declaration indicates wheter a module is a library module. A module declaration consists of the keyword module followed by a namespace prefix and a string literal which must contain a valid URI.The URI identifies the target namespace of the library module, which is the namespace for all variables and functions exported by the library module. The name of every variable and function declared in a library module must have a namespace URI that is the same as the target namespace of the module; otherwise a static error is raised.

Any module may import one or more library modules by means of a module import that specifies the target namespace of the library modules to be imported. When a module imports one or more library modules, the variables and functions declared in the imported modules are added to the static context and (where applicable) to the dynamic context of the importing module.

Namespace declaration

A namespace declaration declares a namespace prefix and associates it with a namespace URI, adding the (prefix, URI) pair to the context. The string literal used in a namespace declaration must be a valid URI, and may not be a zero-length string [err:XQ0053]. The namespace declaration is in scope throughout the query in which it is declared, unless it is overridden by a namespace declaration attribute in an element constructor.

Example: Namespace declaration

declare namespace gael = "http://www.gael.fr";

Multiple declarations of the same namespace prefix in the Prolog result in a static error.[err:XQ0033] However, a declaration of a namespace in the Prolog can override a prefix that has been predeclared in the static context.

In the previous example, the namespace has been bound to the prefix "gael". That prefix may the further be used for variable references, function calls or declaration

or at element construction.

Default Namesapce declaration

Default namespace declarations can be used in a Prolog to facilitate the use of unprefixed QNames. The string literal used in a default namespace declaration must be a valid URI, and may not be a zero-length string.[err:XQ0046] The following kinds of default namespace declarations are supported:

Declaration of a default element/type namespace declares a namespace URI that is associated with unprefixed names of elements and types. If no default element/type namespace is declared, unqualified names of elements and types are in no namespace. The following example illustrates the declaration of a default namespace for elements:

Example:

```
declare default element namespace "http://www.gael.fr/drb";
```

If a direct element constructor includes an attribute named xmlns, it is considered to be a namespace declaration attribute that specifies a new default element/type namespace within the scope of the constructed element and its descendants.

A Prolog may contain a declaration for a default function namespace. If no default function namespace is declared, the default function namespace is the namespace of XPath/XQuery functions, http://www.w3.org/2003/11/xpath-functions. The following example illustrates the declaration of a default function namespace:

Example:

```
declare default function namespace "http://www.esa.int/pds";
```

The effect of declaring a default function namespace is that all functions in the default function namespace, including implicitly-declared constructor functions, are aliased with a name that has the original local name, but no namespace URI. The function may then be called using either its original name or its alias-that is, the namespace prefix becomes optional. When a function call uses a function name with no prefix, the local name of the function must match a function (including implicitly-declared constructor functions) in the default function namespace.

Base URI declaration

A base URI declaration specifies the base URI property of the static context, which is used when resolving relative URIs within a module. A static error [err:XQ0032] is raised if more than one base URI declaration is found in a query prolog

Chapter 4: Primary Expressions

Primary expressions are the basic primitives of the language.

```
PrimaryExpr ::=  Variable
               | Literal
               | FunctionCall
               | ParenthesizedExpr
```

Literals

A literal is a direct syntactic representation of a simple value. The DRB query engine supports two kinds of literals: string literals and numeric literals.

```
Literal        ::= NumericLiteral |  StringLiteral
NumericLiteral ::= IntergeLiteral | DoubleLiteral
IntergeLiteral ::= [0-9]+
DoubleLiteral  ::= [0-9]+ ("." [0-9]+)? (("e" | "E")? ("+" | "-")? [0-9]+)?
StringLiteral  ::=  ("\"" (~["\""])* "\"") | ("'" (~["'"])* "'")>
```

The value of a string literal is a singleton sequence containing an item whose primitive type is "string" and whose value is the string denoted by the characters between the delimiting quotation marks.

Note: In the definition of the StringLiteral production, we remind that the BNF production ~["\""] matches any character but a double quote and ~["'"] matches any character but a quote.

The value of a numeric literal containing no "." and no "e" or "E" character is a singleton sequence containing an item whose type is "integer" and whose value is obtained by parsing the numeric literal according to the rules of the "integer" The value of a numeric literal containing ".", "e" or "E" character is a singleton sequence containing an item whose primitive type is "double" and whose value is obtained by parsing the numeric literal according to the rules of the "double" data type.

Example: Literal expressions

"12.5" denotes the string containing the chars '1', '2', '.', and '5'.

12 denotes the integer value twelve.

12.5 denotes the double value twelve and one half.

123e2 denotes the double value twelve thousand, five hundred.

Note: The DRB query engines does not distinguish the DecimalLiteral from the DoubleLiteral productions as specified by the W3C. They are both considered as DoubleLiteral productions.

Note: The DRB query engines does not supports double literals without integer part (i.e. starting with a dot "."). A set of zeros must be explicitly written in the query. This is a restriction from the XQuery specifications.

Variables

A variable evaluates to the value to which the variable's name is bound in the evaluation context. It is an error if the variable's name is not defined in the evaluation context. Variables can be bound by clauses in For or Let expressions and by function calls which bind values to the formal parameters of functions.

```
Variable ::= "$" ([a-Z] | "_") ([a-Z] | [0-9] | "." | "-" | "_")*
```

The binding of a variable in an expression always overrides any in-scope binding of a variable with the same name. For example, for $v in (1,2) return for $v in (3,4) return $v evaluates to (3, 4, 3, 4) because the definition of $v in the inner for expression overrides the definition of $v in the outer for expression.

Use of variable

```
{-- Use of variables --}
let $product := GOM_NL__2P/(schemas/gom/GOM_NL__2P.xml)GOM_NL__2P
for $tgtrecord in $product/tangent_mds/tangent_mdsr
return $tgtrecord/air
```

Note: The Variable name definition is slightly modified from the XQuery specifications.

The enabled character set has been reduced.

Parenthesized Expressions

Parentheses may be used to enforce a particular evaluation order in expressions that contain multiple operators, as shown in the following example:

Example: Parenthesized expression

The expression (2 + 4) * 5 evaluates to thirty; the parenthesized expression (2 + 4) is evaluated first and its result is multiplied by five. Without parentheses, the expression 2 + 4 * 5 evaluates to twenty-two, because the multiplication operator has higher precedence than the addition operator.

Function Calls

A function call consists of a function name followed by a parenthesised list of zero or more expressions. The function name must match the name of a function listed in the next chapter. The expressions between the parentheses provide the arguments of the function call. The number of arguments must equal the number of formal arguments specified in the function's signature; otherwise an error is raised.

Because the functions are under definition and still vary significantly between two issues of the working draft, the DRB XQuery engine does not already implement the overall built-in functions. However, a set of important and stable functions has already been implemented. The next sections provide a complete list of these functions, accompanied by the extended built-in functions.

```
FunctionCall ::= FunctionName "(" (Expr ( "," Expr )* )? ")"

FunctionName ::= ([a-Z] | "_") ([a-Z] | [0-9] | "." | "-" | "_")*
```

Note: The prefix of the function (e.g. math in math:power()) is not mandatory as far as the functions can be identified without ambiguity.

Comments

Comments can be used to annotate a query with information. Comments are

lexical constructs only, and have no meaning within the query.

```
ExprComment  ::= "(:" (<Char>* ":)"
IgnoredChars ::=  (" " | "\t" | "\n" | "\r" )+
```

Comments

```
(: This is an example of comment :)
(: This is an example of (: nested :) comments :)
```

In addition to comments the sequences of IgnoredChars are discarded while the query is parsed. They are separators from the lexical units of the XQuery language (i.e. basically the words composing the query).

Note: For compatibilty reasons, the old comment syntax is still supported (i.e. "{--" and "--}" tokens). However, a warning message is printed out once in the logger system, inviting to modify the evaluated script.

Chapter 5: Path Expressions

A path expression locates nodes within a tree, and returns a sequence of distinct nodes in document order. A path expression is always evaluated with respect to an evaluation context. There are two kinds of path expressions: relative and absolute.

```
PathExpr          ::= AbsolutePathExpr | RelativePathExpr
AbsolutePathExpr ::= "/" RelativePathExpr
RelativePathExpr ::= GeneralStep ( "/" GeneralStep )* "/"?
```

Relative Path Expression

A relative path expression consists of one or more steps separated by /. The steps in a relative path expression are composed together from left to right. Each step in turn selects a sequence of nodes. An initial sequence of steps is composed together with a following step as follows: each node selected by the initial sequence is used to provide a focus for the following step (c.f. Evaluation Context). The following step is executed for each focus, and the resulting sequences of nodes are merged, preserving input data source order. For example, div1/para selects the para element children of the div1 element children of the context node, or, in other words, the para element grandchildren that have div1 parents.

Caution: The PathExpr has been slightly modified from the W3C specifications to allow an optional "/" at the end of the path expressions. This does not alter the meaning of the path and is closer to the familiar syntax of file system paths.

Absolute Path Expression

An absolute path expression consists of / followed by a relative path expression. A / by itself selects the root node containing the context node.

Note: In the framework of the W3C, only XML documents are considered. The DRB implementation opens the XQuery's to any data sources including the file system. The root node containing the context node is therefore in most of cases the file system root. This functionality is one of the most interesting of using the DRB API. Thanks to DRB it possible to query file systems or structured files as well as XML documents. It is foreseen to make the DRB to support many other data sources such as HTTP, ZIP archives as well as databases (e.g. RDMS such as Oracle and Sybase).

Caution: The advantage of querying many data sources including the file system may lead to heavy query evaluations. The queries shall therefore be written with extreme caution before their execution.

If it is followed by a relative path expression, then the path expression selects the sequence of nodes that would be selected by the relative path expression relative to the root node of the context node.

Note: The following productions are not already supported by the current DRB query engine implementation: "//", AxisStep. In addition only the abbreviated syntax of the path expression is implemented.

General Steps

```
GeneralStep   ::= StepParameter? ( ".." | "." | "@" Name | Name ) StepQualifiers
StepParameter ::= "(" Expr ( "," Expr )* ")"
Name       ::= ([a-Z] | "_") ([a-Z] | [0-9] | "." | "-" | "_")*
```

A general step is more or less a primary expression, possibly followed by one or more step qualifiers.

The expression in the general step must evaluate to a sequence of nodes. The result of the general step is this sequence of nodes, modified by any step qualifiers If the expression in the general step returns any values that are not nodes, an error is raised.

As in the case of an axis step, a general step is executed once if it is the first step in the path expression. If it is not the first step, the general step is executed once for each node selected by the preceding step, using the focus provided by each of these nodes in turn, as described in the section named "Evaluation Context". The nodes selected by the general step are then used in turn to provide focuses for execution of the following step, if any.

In most of cases the general step are reduced to the name of the nodes to be selected (i.e. Name production), ".." or ".". Where "." stands for the context node and ".." for the parent of the context node.

If the name of the step is prefixed by a "@", the step is to be considered as an attribute step. It therefore selects the corresponding attribute in the context node.

Step Qualifiers

Step qualifiers are predicates which are used to filter a node sequence by applying some test.

```
StepQualifiers ::= ( "[" Expr "]" )*
```

A predicate consists of an expression, called a predicate expression, enclosed in square brackets. A predicate serves to filter a node sequence, retaining some nodes and discarding others. For each node in the node sequence to be filtered, the predicate expression is evaluated using a focus derived from that node, as described in the section named "Evaluation Context". The result of the predicate expression is coerced to a Boolean value, called the predicate truth value, as described below. Those nodes for which the predicate truth value is true are retained, and those for which the predicate truth value is false are discarded.

The predicate truth value is derived by applying the following rules, in order:

If the value of the predicate expression is an empty sequence, the predicate truth value is false.

If the value of the predicate expression is one simple value of a numeric type, the value is rounded to an integer using the rules of the round function, and the predicate truth value is true if and only if the resulting integer is equal to the value of the context position.

If the value of the predicate expression is one simple value of type Boolean, the predicate truth value is equal to the value of the predicate expression.

If the value of the predicate expression is a sequence containing at least one node, the predicate truth value is true. Note that the predicate truth value in this case does not depend on the content of the node.

In any other case, an error is raised.

Range constraints

```
let $product := GOM_NL__2P/(schemas/gom/GOM_NL__2P.xml)GOM_NL__2P

for $tgtrecord in $product/tangent_mds[1]/tangent_mdsr[2 to 8]

return
   <rec>
      <pos>
         position()
      </pos>
      <air>
         $tgtrecord/air
      </air>
   </rec>
```

Note: The current implementation of the DRB query engine does not supports the dereferences mechanism defined by the W3C.

Step Parameters

Caution: The step parameters are not defined in the XQuery specifications. Because the XQuery language has been initially designed for XML node and because the access to XML nodes does not require parameter it was not required by the W3C to provide parameters for this purpose. On the contrary, on many other data sources some parameters may be required. For instance while opening a database, a HTTP connection it may be necessary to provide a user name and password. In the current implementation the unique step parameter is the XML schema file that has to be used to open a file node as a structured data file.

Chapter 6: Arithmetic Expressions

XQuery provides arithmetic operators for addition, substraction, multiplication, division, and modulus, in their usual binary and unary forms.

```
AdditiveExpr       ::= Expr ("+" | "-") Expr
MultiplicativeExpr ::= Expr ("*" | "div" | "mod") Expr
UnaryExpr          ::= "-" Expr
```

The binary subtraction operator must be preceded by white space if it follows variable, in order to distinguish it from a hyphen, which is a valid name character. For example, $a-b will be interpreted as a single word.

The operands of an arithmetic expression do not have required types. An arithmetic expression is evaluated by applying the following rules, in order, until an error is raised or a value is computed:

- If any operand is a sequence of length greater than one, a type exception is raised.
- If any operand is an empty sequence, the result is an empty sequence.
- If any operand is a node, its typed value extracted. If the node has no typed value, or if the typed value is a sequence of more than one value, an error is raised. If the typed value returns the empty sequence, the result of the expression is the empty sequence.
- If the arithmetic expression has two numeric operands of different types, one of the operands is promoted to the type of the other operand, following the promotion rules defined in the chapter named "Data definition with XML-Schema". For example, an integer value be promoted to double.
- If the operand type(s) are valid for the given operator, the operator is applied to the operand(s), resulting in either a simple value or an error (for example, an error might result from dividing by zero). The combinations of simple types that are accepted by the various arithmetic operators, and their respective result types, are listed in the chapter named "Data definition with XML-Schema". If the operand type(s) are not valid for the given operator, a type exception is raised.

Note: The case of non typed values described in the XQuery language specifications never occurs with DRB because all the value are strongly typed. In addition the unary plus is not supported in the current implementation of the DRB query engine.

Chapter 7: Sequence Expressions

The present chapter addresses how sequence can be created or manipulated in XQuery.

Simple sequence constructions

One way to construct a sequence is by using the comma operator, which evaluates each of its operands and concatenates the resulting sequences, in order, into a single result sequence. In addition, empty parentheses can be used to denote an empty sequence.

A sequence may contain duplicate atomic values or nodes, but a sequence is never an item in another sequence. When a new sequence is created by concatenating two or more input sequences, the new sequence contains all the items of the input sequences and its length is the sum of the lengths of the input sequences.

Examples of sequence constructions

```
10, 1, 2, 3, 4 (: a simple sequence of integers :)
() (: the empty sequence :)
1, 2, () , 3 (: results to the 1, 2, 3 sequence :)
book, author (: a sequence of nodes :)
1+2, 3 (: a sequence resulting to 3, 3 :)
$var1, $var2 (: a sequence of variables :)
1, $var, book (: a mixed sequence :)
```

Range expression

A range expression can be used to construct a sequence of consecutive "integers". Each of the operands of the to operator is converted as though it was an argument of a function with the expected parameter type xs:integer?. If either operand is an empty sequence, or if the integer derived from the first operand is greater than the integer derived from the second operand, the result of the range expression is an empty sequence. Otherwise, the result is a sequence containing the two integer operands and every integer between the two operands, in increasing order.

Examples of sequences constructed from a range

```
1 to 5 (: equivalent to 1, 2, 3, 4, 5 :)
```

```
10 to 10 (: construct the singleton sequence 10 :)

5 to 1 (: bad order results to the empty sequence () :)

1, 2 to 4, 5 (: mixed construction => 1, 2, 3, 4, 5 :)

$var to 5 (: operand from a variable :)

$var to my:func() (: operand from a function :)

(1+3) to my:func() (: computed left operand :)
```

Combining sequences of nodes

One may combine sequences of "nodes" using one of the following combining operators:

- **union or '|':** The union and | operators are equivalent. They take two node sequences as operands and return a sequence containing all the nodes that occur in either of the operands.
- **intersect:** The intersect operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in both operands.
- **except:** The except operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in the first operand but not in the second operand.

All these operators eliminate duplicate nodes from their result sequences based on node "identity".

Example of combinations

```
(: In the following examples it is assumed that:

   $seq1 is bound to (A, B)
   $seq2 is bound to (A, B)
   $seq3 is bound to (B, C)

   where all A are identical, all B are identical and all C
   are identical.
:)

$seq1 union $seq2 (: evaluates to the sequence (A, B) :)

$seq2 union $seq3 (: evaluates to the sequence (A, B, C). :)

$seq1 intersect $seq2 (: evaluates to the sequence
                         (A, B). :)

$seq2 intersect $seq3 (: evaluates to the sequence
                         containing B only. :)

$seq1 except $seq2 (: evaluates to the empty sequence. :)

$seq2 except $seq3 (: evaluates to the sequence
                      containing A only. :)
```

Caution: Node identity It shall always be considered that two nodes are "identical" if and only if they are corresponding to the same instance. Two nodes with the same name, the same value, etc may not "identical". Check carefully the following examples that illustrate this potential confusion.

Example of potential confusions

```
<a/> union <a/> (: results to <a/>, <a/> because although
                   both nodes have the same name and value,
                   they are not the same :)

doc("book.xml")/book union doc("book.xml")/book

                (: is equivalent to the previous case because
                   altough both book are originating from
                   the same document, they are not the same
                   nodes :)

let $book := doc("book.xml")/book
return
  $book union $book

                (: This time results to $book only because
                   the variable bind the "same" node :)
```

Chapter 8: Comparison Expressions

General comparisons are defined by adding existential semantics to a value comparisons. The operands of a general comparison may be sequences of any length. The result of a general comparison is always true or false.

```
GeneralComp ::= Expr ("=" | "!=" | "<" " " | "<=" | ">" | ">=")  Expr
```

Note: The "<" comparison operator must be followed by white space in order to distinguish it from a tag-open character.

Note: The ValueComp, NodeComp and OrderComp defined in the XQuery specifications are not supported by the current implementation of DRB query engine.

The general comparison A = B is true for sequences A and B if the value comparison a eq b is true for some item a in A and some item b in B. Otherwise, A = B is false.

Similarly:

A != B is true if and only if a not equals b is true for some a in A and some b in B.

A < B is true if and only if a less than b is true for some a in A and some b in B.

A <= B is true if and only if a less or equals b is true for some a in A and some b in B.

A > B is true if and only if a greater than b is true for some a in A and some b in B.

A >= B is true if and only if a greater or equals b is true for some a in A and some b in B.

Chapter 9: Logical Expressions

A logical expression is either an and-expression or an or-expression. The value of a logical expression is always one of the Boolean values true or false.

```
OrExpr  ::= Expr "or" Expr

AndExpr ::= Expr "And  Expr
```

The first step in evaluating a logical expression is to reduce each of its operands to an effective Boolean value by applying the following rules, in order:

If the operand is an empty sequence, its effective Boolean value is false.

If the operand is one simple Boolean value, the operand serves as its own effective Boolean value.

If the operand is a sequence that contains at least one node, its effective Boolean value is true.

In any other case, an error is raised.

An and-expression returns the value true if the effective Boolean values of both of its operands are true; otherwise it returns the value false.

An or-expression returns the value false if the effective Boolean values of both of its operands are false; otherwise it returns the value true.

Note: As already mentioned in the present document, the current implementation of DRB query engine does not support the "not()" function specified in the XQuery language.

Chapter 10: Element Constructors

An element constructor creates a node. If the name, attributes, and content of the element are all constants, the element constructor uses standard XML notation.

In an element constructor, the name used in an end tag must match the name of the corresponding start tag.

In an element constructor, curly braces { } delimit enclosed expressions, distinguishing them from literal text. Enclosed expressions are evaluated and replaced by their value, whereas material outside curly braces is simply treated as literal text.

The previous example is equivalent to the former one apart that "isbn" attribute and the name of "Henry" have been computed. An unlimited number of enclosed expressions can be used in the attribute or node values.

It is time now to remind the example present in the introduction chapter, where intent to illustrate how simple it is to produce dynamically an output document. Here the output language is XHTML but it is reminded that DRB may produce a lerge variety of other formats.

Chapter 11: FLWOR Expressions

XQuery provides a FLWOR expression for iteration and for binding variables to intermediate results. This kind of expression is often useful for computing joins between two or more documents and for restructuring data. The name "FLWOR", pronounced "flower", stands for the keywords for, let, where, and return, the four clauses found in a FLWOR expression.

FLWOR Expression

```
FLWORExpr    ::= (ForClause |  LetClause)+ WhereClause? "return" Expr
ForClause   ::= "for" Variable "in" Expr ("," Variable "in" Expr)*
LetClause   ::= "let" Variable ":=" Expr ("," Variable ":=" Expr)*
WhereClause ::= "where" Expr
```

The clauses of a FLWOR Expression are interpreted as follows:

A for clause associates one or more variables with expressions, creating tuples of variable bindings drawn from the Cartesian product of the sequences of values to which the expressions evaluate. The variable binding tuples are generated as an ordered sequence as described below.

A let clause binds a variable directly to an entire expression. If for clauses are present, the variable bindings created by let clauses are added to the tuples generated by the for clauses. If there are no for clauses, the let clauses generate one tuple with all variable bindings.

A where clause can be used as a filter for the tuples of variable bindings generated by the for and let clauses. The expression in the where clause, called the where-expression, is evaluated once for each of these tuples. If the effective Boolean value of the where-expression is true, the tuple is retained and its variable bindings are used in an execution of the return clause. If the effective Boolean value of the where-expression is false, the tuple is discarded. The effective Boolean value of the where-expression is computed by applying the following rules, in order:

  # If the where-expression returns an empty sequence, its effective Boolean value is false.

  # If the where-expression returns one simple Boolean value, this value serves as the effective Boolean value.

  # If the where-expression returns a sequence that contains at least one node, its effective Boolean value is true.

  # In any other case, an error is raised.

- The return clause contains an expression that is used to construct the result of the FLWOR expression. The return clause is invoked once for every tuple generated by the for and let clauses, after eliminating any tuples that do not satisfy the conditions of a where clause. The expression in the return clause is evaluated once for every invocation, and the result of the FLWOR expression is an ordered sequence containing the results of these invocations.

A variable name may not be used before it is bound, nor may it be used in the expression to which it is bound. Any variable bound in a for or let clause is in scope until the end of the FLWOR expression in which it is bound. If the variable name used in the binding was already bound in the current scope, the variable name refers to the newly bound variable until that variable goes out of scope. At this point, the variable name again refers to the variable of the prior binding.

Although for and let both bind variables, the manner in which variables are bound is quite different. In a let clause, the variable is bound directly to the expression, and it is bound to the expression as a whole.

Chapter 12: Conditional Expressions

XQuery provides a conditional expression based on the keywords if, then, and else.

```
IfExpr ::= "if" "(" Expr ")" "then" Expr "else" Expr
```

The expression following the if keyword is called the test expression, and the expressions following the then and else keywords are called the then-expression and else-expression, respectively.

The first step in processing a conditional expression is to find the effective Boolean value of the test expression, defined as follows:

- If the sequence is empty, its effective Boolean value is false.
- If the sequence contains a single atomic value of type xs:boolean, this value is the effective Boolean value of the sequence.
- If the sequence contains at least one node, its effective Boolean value is true.
- In any other case, the effective Boolean value is false

Note: The last case of the list does not complies with the XQuery specifications. The current implementation of the XQuery engine does not already supports the management of errors. It is therefore necessary to have a false or true result and no other value. The false result has been preferred because it is closed to the first case.

The value of a conditional expression is defined as follows: If the effective Boolean value of the test expression is true, the value of the then-expression is returned. If the effective Boolean value of the test expression is false, the value of the else-expression is returned.

Chapter 13: Functions and Operators

This chapter defines basic operators and functions supported by the DRB XQuery implementation.

Accessors

fn:node-name()

This function returns the name of the node as a string.

```
fn:node-name($node as node()?) as xs:string
```

In the DRB implementation, this function returns exactly the same value as the fn:name function, apart that, if no parameter is provided, the context item is not considered and the empty sequence is returned.

fn:nilled()

Caution: Not supported by the current implementation.

fn:string()

Caution: Not supported by the current implementation.

fn:data()

Returns a sequence of atomic values extracted from the input sequence parameter.

```
fn:data($arg as item()*) as xdt:anyAtomicType*
```

The returned sequence is composed of zero or more atomic values produced by applaying the following rules to each item of the input sequence:

If the item is an atomic value : the item is returned.

If the item is a node or an attribute : the value of the node or the

attribute is returned (its may result of a sequence of more than one item).

Simple query using the data() built-in function

```
let $var1 := xs:int(15)
let $var2 := xs:int(55)
return
<result>
   <first>{data($var1)}</first>
   <first>{$var2}</first>
</result>
```

This query extracts the node values using the data() built-in function

```
let $product := ../products/MER_RR__1P.N1/
   (../schemas/int/esa/ENVISAT/MER/MER_RR__1P.xsd)MER_RR__1P

for $element in $product/mph/*
return
<result>

   <value>
   {
      data($element)
   }
   </value>
</result>
```

fn:base-uri()

Caution: Not supported by the current implementation.

fn:document-uri()

Caution: Not supported by the current implementation.

The Error Function

fn:error()

Interrupts the dynamic evaluation of the query by raising an error.

```
fn:error($message as xdt:anyAtomicType?) as none
```

If a parameter is provided, it is converted to an xs:string and appended to the error message printed out in the logger system as an ERROR message.

This function is useful to control the evaluation of an XQuery. As an example, it may be used for verifying a function parameter or for controlling the value af an external variable prior to execute an expression.

It my also be used for debugging a piece of script to avoid lagging processing outside the tested area. In this way, fn:error is complementary to the fn:trace function.

The Trace Function

fn:trace()

This function is intended to be used in debugging queries by providing a trace of their execution.

```
fn:trace($value as item()*,
         $label as xs:string) as item()*
```

The input $value is returned, unchanged, as the result of the function. In addition, the inputs $value, converted to an xs:string, and $label are directed to a trace data set.

The output label is printed out to the logger system as WARN typed message.

The order of invocations of the fn:trace function depends on the query script. It may vary according to the optimisation and planification phases.

Constructor Functions

Constructor Functions for XML Schema Built-in Types

The constructor functions are dedicated to create atomic values from another one usually of a different type. They may be compared to cast expressions converting the types an atomic values. They are useful for constructing types different from the literal expressions (i.e. string, integer, decimal or double) natively supported by the XQuery language.

Example of a boolean constructed from a string

```
xs:boolean("true")
```

Example of a durations constructed from a string

```
xs:duration("P10S") (: a duration of 10 seconds :)
xs:duration("PT1H20M") (: a duration an hour and twenty minutes :)
```

Casting a variable of any type to an integer value

```
xs:integer($var)
```

The form of that Constructor Function for a type prefix:TYPE is:

```
prefix:TYPE($arg as xdt:anyAtomicType) as prefix:TYPE?
```

If $arg is the empty sequence, the empty sequence is returned. For example, the signature of the constructor function corresponding to the xs:unsignedInt type defined in the schema part is:

Caution: Unsupported behavior The current implementation does not support constructor function calls with no parameter. It is therefore not already possible to create an atomic type with is default value.

```
xs:unsignedInt($arg as xdt:anyAtomicType) as xs:unsignedInt?
```

Invoking the constructor function xs:unsignedInt(12) or xs:unsignedInt("12") returns the xs:unsignedInt value 12. The same result would also be returned if the constructor function were to be invoked with a node that had a typed value equal to the xs:unsignedInt 12.

The reference list of the supported type and constructors is the chapter "XML Shemas" of the part "Schemas of data sources". This chapter defines the suppported types and in particular their name that controls the constructor function name and prefix (i.e. prefix:TYPE mentioned above) and the possible value domains.

Nevertheless, for information, the supported constructor functions for the built-in types are listed below with accompanying exampes:

xs:string:

Function signature

```
xs:string($arg as xdt:anyAtomicType) as xs:string?
```

Examples

```
xs:string("hello") (: from a string :)
xs:string(100) (: from an integer: returns "100" :)
xs:string(100) (: from an integer: returns "100" :)
xs:string(1 > 2) (: from a comparison expression resulting
                    to a boolean value: returns "false" :)
```

xs:boolean:

Function signature

```
xs:boolean($arg as xdt:anyAtomicType) as xs:boolean?
```

Examples

```
xs:boolean("hello") (: from a string :)
xs:boolean("1") (: from another string: returns true :)
xs:boolean(0)   (: from an integer: returns false :)
```

xs:decimal:

Function signature

```
xs:decimal($arg as xdt:anyAtomicType) as xs:decimal?
```

Examples

```
xs:decimal("-1.2") (: from a string : build -1.2 decimal
                      value :)
xs:decimal(-1.2)   (: from an integer : build -1.99999
                      double value :)
```

xs:float:

Function signature

```
xs:float($arg as xdt:anyAtomicType) as xs:float?
```

Examples

```
xs:float("-1.2") (: from a string: returns -1.9999999 :)
xs:float(2)      (: from an integer: returns 2.0 :)
```

xs:double:

Function signature

```
xs:double($arg as xdt:anyAtomicType) as xs:double?
```

Examples

```
xs:double("-1.2") (: from a string: returns -1.9999999 :)
xs:double(2)      (: from an integer: returns 2.0 :)
```

xs:duration:

Function signature

```
xs:duration($arg as xdt:anyAtomicType) as xs:duration?
```

Note: The xs:duration is usually constructed from a string. The string constructor should follow the ISO-8601 standard also repeated by the W3C XML Schema recommendation. This standard may be summarized with the following example "-P1Y2M3DT10H30M20.12S" that corresponds to a negative duration of 1 year, 2 months, 3 days, 10 hours, 30 minutes and 20.12 seconds. Almost all couples of value-unit (e.g. 2M or 10H) are optional but at least one shall be provided. The "T" symbol separates the date and time components of the duration. The designator "T" must be absent if and only if all of the time items are absent. The designator "P" must always be present. The heading minus is optional.

Examples

```
(: a duration of 1 year, 2 months, 3 days, 10 hours,
   and 30 minutes :)

xs:duration("P1Y2M3DT10H30M") (: a duration of 1 year,

xs:duration("P10Y) (: A druation of 10 years :)
xs:duration("PT20S) (: A druation of 20 seconds :)
```

xs:dateTime:

Function signature

```
xs:dateTime($arg as xdt:anyAtomicType) as xs:dateTime?
```

Caution: Conversly to the XQuery working draft, the construction of an xs:dateTime from a string shall not conform to the ISO-8601 but the ASCII form of European Space Agency (ESA) ENVISAT MJD date defined in the MGA-GS-2009 ENVISAT PDS Format Specification, Annex. An explicit example of this format may be "01-JAN-2005 00:00:00.000000". All characters shall always be expressed.

Examples

```
(: first January 2005 at noon :)

xs:dateTime("01-JAN-2005 00:00:00.000000")
```

xs:date():

Function signature

```
xs:dateTime($arg as xdt:anyAtomicType) as xs:dateTime?
```

Caution: Conversly to the XQuery working draft, the construction of an xs:date from a string shall not conform to the ISO-8601 but the ASCII form of European Space Agency (ESA) ENVISAT MJD date defined in the MGA-GS-2009 ENVISAT PDS Format Specification, Annex. An explicit example of this format may be "01-JAN-2005 00:00:00.000000". All characters shall always be expressed.

Examples

```
(: first January 2005 at noon :)

xs:date("01-JAN-2005 00:00:00.000000")
```

xs:anyURI:

Function signature

```
xs:anyURI($arg as xdt:anyAtomicType) as xs:anyURI?
```

Examples

```
xs:string("http://www.gael.fr")
```

xs:integer:

Function signature

```
xs:integer($arg as xdt:anyAtomicType) as xs:integer?
```

Examples

```
xs:string("-100")
xs:string(-100)
xs:string(2.3) (: returns 2 :)
```

xs:nonPositiveInteger:

Function signature

```
xs:nonPositiveInteger($arg as xdt:anyAtomicType)
   as xs:nonPositiveInteger?
```

Caution: The xs:nonPositiveInteger is currently implemented as an xs:integer at construction.

Examples

```
xs:nonPositiveInteger("-1") (: returns -1 :)
xs:nonPositiveInteger("1") (: returns 1 :)
```

xs:negativeInteger:

Function signature

```
xs:negativeInteger($arg as xdt:anyAtomicType)
   as xs:negativeInteger?
```

Caution: The xs:negativeInteger is currently implemented as an xs:integer at construction.

Examples

```
xs:negativeInteger("-1") (: returns -1 :)
xs:negativeInteger("1") (: returns 1 :)
```

xs:long:

Function signature

```
xs:long($arg as xdt:anyAtomicType) as xs:long?
```

Examples

```
xs:long("1904")
xs:long("-1904") (: return -1904 :)
```

xs:int:

Function signature

```
xs:int($arg as xdt:anyAtomicType) as xs:int?
```

Examples

```
xs:int("1") (: returns 1 :)
xs:int(-1) (: returns -1 :)
```

xs:short:

Function signature

```
xs:short($arg as xdt:anyAtomicType) as xs:short?
```

Examples

```
xs:short("3") (: 3 :)
xs:short("-3") (: -3 :)
xs:short("3000000") (: dynamic error :)
```

xs:byte:

Function signature

```
xs:byte($arg as xdt:anyAtomicType) as xs:byte?
```

Examples

```
xs:byte("3") (: 3 :)
xs:byte("-3") (: -3 :)
xs:byte("3000000") (: dynamic error :)
```

xs:nonNegativeInteger:

Function signature

```
xs:nonNegativeInteger($arg as xdt:anyAtomicType)
   as xs:nonNegativeInteger?
```

Caution: The xs:nonNegativeInteger is currently implemented as an xs:integer at construction.

Examples

```
xs:nonNegativeInteger("3") (: 3 :)
xs:nonNegativeInteger("-3") (: -3 :)
xs:nonNegativeInteger("3000000") (: 3000000 :)
```

xs:unsignedLong:

Function signature

```
xs:unsignedLong($arg as xdt:anyAtomicType)
   as xs:unsignedLong?
```

Examples

```
xs:unsignedLong("3") (: 3 :)
xs:unsignedLong("-3") (: 18446744073709551613 :)
xs:unsignedLong("3000000") (: 3000000 :)
```

xs:unsignedInt:

Function signature

```
xs:unsignedInt($arg as xdt:anyAtomicType)
   as xs:unsignedInt?
```

Examples

```
xs:unsignedInt("3") (: 3 :)
```

```
xs:unsignedInt("-3") (: 4294967293 :)
xs:unsignedInt("3000000") (: 3000000 :)
```

unsignedShort:

Function signature

```
xs:unsignedShort($arg as xdt:anyAtomicType)
   as xs:unsignedShort?
```

Examples

```
xs:unsignedShort("3") (: 3 :)
xs:unsignedShort("-3") (: 65533 :)
xs:unsignedShort("3000000") (: dynamic error :)
```

xs:unsignedByte:

Function signature

```
xs:unsignedByte($arg as xdt:anyAtomicType)
   as xs:unsignedByte?
```

Examples

```
xs:unsignedByte("3") (: 3 :)
xs:unsignedByte("-3") (: 253 :)
xs:unsignedByte("3000000") (: dynamic error :)
```

xs:positiveInteger:

Function signature

```
xs:positiveInteger($arg as xdt:anyAtomicType)
   as xs:positiveInteger?
```

Caution: The xs:positiveInteger is currently implemented as an xs:integer at construction.

Examples

```
xs:positiveInteger("3") (: 3 :)
xs:positiveInteger("-3") (: -3 :)
xs:positiveInteger("3000000") (: 3000000 :)
```

Constructor Functions for User-Defined Types

Constructor functions are not supported in the current implementation.

Functions and Operators on Numerics

Operators on Numeric Values

op:numeric-add($arg1 as numeric, $arg2 as numeric) as numeric

Caution: This function is not supported.

op:numeric-subtract($arg1 as numeric, $arg2 as numeric) as numeric

Caution: This function is not supported.

op:numeric-multiply($arg1 as numeric, $arg2 as numeric) as numeric

Caution: This function is not supported.

op:numeric-divide($arg1 as numeric, $arg2 as numeric) as numeric

Caution: This function is not supported.

op:numeric-integer-divide($arg1 as xs:numeric, $arg2 as xs:numeric) as xs:integer

Caution: This function is not supported.

op:numeric-mod($arg1 as numeric, $arg2 as numeric) as numeric

Caution: This function is not supported.

op:numeric-unary-plus($arg as numeric) as numeric

> **Caution:** This function is not supported.

op:numeric-unary-minus($arg as numeric) as numeric

> **Caution:** This function is not supported.

Comparison of Numeric Values

This section defines the following comparison operators on numeric values.

op:numeric-equal($arg1 as numeric, $arg2 as numeric) as xs:boolean

> **Caution:** This function is not supported.

op:numeric-less-than($arg1 as numeric, $arg2 as numeric) as xs:boolean

> **Caution:** This function is not supported.

op:numeric-greater-than($arg1 as numeric, $arg2 as numeric) as xs:boolean

> **Caution:** This function is not supported.

Functions on Numeric Values

The following functions are defined on numeric types. Each function returns a value of the same type as the type of its argument.

fn:abs

> **Caution:** This function is not supported.

fn:ceiling

> **Caution:** This function is not supported.

fn:floor

Caution: This function is not supported.

fn:round

Caution: This function is not supported.

fn:round-half-to-even

Caution: This function is not supported.

Functions on Strings

This section discusses functions and operators on the xs:string datatype and the datatypes derived from it.

Functions to Assemble and Disassemble Strings

fn:codepoints-to-string($arg as xs:integer*) as xs:string

Caution: This function is not supported.

fn:string-to-codepoints($arg as xs:string?) as xs:integer*

Caution: This function is not supported.

Equality and Comparison of Strings

fn:compare($comparand1 as xs:string?, $comparand2 as xs:string?) as xs:integer?

Caution: This function is not supported.

fn:compare($comparand1 as xs:string?, $comparand2 as xs:string?, $collation as xs:string) as xs:integer?

Caution: This function is not supported.

Functions on String Values

The following functions are defined on values of type xs:string and types derived from it.

fn:concat

Concatenates two or more xs:strings.

```
xs:string fn:concat() as xs:string,

xs:string fn:concat($op1 as xs:string?) as xs:string,

xs:string fn:concat($op1 as xs:string?,
                    $op2 as xs:string?) as xs:string,
```

Accepts zero or more xs:strings as arguments. Returns the xs:string that is the concatenation of the values of its arguments. The resulting xs:string might not be normalized according to any Unicode or W3C normalization form. If called with no arguments, returns the zero-length string. If any of the arguments is the empty sequence, it is treated as the zero-length string.

The concat() function is specified to allow an arbitrary number of xs:string arguments that are concatenated together.

fn:string-join

Caution: This function is not supported.

fn:substring

Returns the xs:string located at a specified place in an xs:string. If the value of $sourceString is the empty sequence, the empty sequence is returned.

```
xs:string fn:substring($source_string as xs:string,
                       $startingLoc   as xs:double)

xs:string fn:substring($source_string as xs:string,
                       $startingLoc   as xs:double,
                       $length        as xs:double)
```

Otherwise, returns the portion of the value of $sourceString

beginning at the position indicated by the value of $startingLoc and continuing for the number of characters indicated by the value of $length. More specifically, returns the characters in $sourceString whose position $p obeys: round($startingLoc) <= $p < round($startingLoc) + round($length).

If $startingLoc is zero or negative, the substring includes characters from the beginning of the $sourceString.

If $length is not specified, the substring includes characters to the end of $sourceString.

If $length is greater than the number of characters in the value of $sourceString following $startingLoc, the substring includes characters to the end of $sourceString.

The first character of a string is located at position 1, not position 0.

```
substring("abcdefghijkl",2) returns "bcdefghijkl"
substring("abcdefghijkl",2,3) returns "bcd"
```

fn:string-length

Caution: This function is not supported.

fn:normalize-space

Caution: This function is not supported.

fn:normalize-unicode

Caution: This function is not supported.

fn:upper-case

Caution: This function is not supported.

fn:lower-case

> **Caution:** This function is not supported.

fn:translate

> **Caution:** This function is not supported.

fn:escape-uri

> **Caution:** This function is not supported.

Functions Based on Substring Matching

The functions described in the section examine a string $arg1 to see whether it contains another string $arg2 as a substring. The result depends on whether $arg2 is a substring of $arg1, and if so, on the range of characters in $arg1 which $arg2 matches.

fn:contains

> **Caution:** This function is not supported.

fn:starts-with

> **Caution:** This function is not supported.

fn:ends-with

> **Caution:** This function is not supported.

fn:substring-before

Returns a substring of a string that precedes in a given string.

```
xs:string fn:substring-before($source_string as xs:string,
                              $bound_string   as xs:string)
```

Same behavior as the fn:substring function with the starting location set to the beginning of the string and the length is determined by a

bounding substring.

If the value of $source_string or $bound_string is the empty sequence, it is interpreted as the zero-length string.

If the value of $bound_string is the zero-length string, then the function returns the zero-length string.

If the value of $source_string does not contain a string that is equal to the value of $bound_string, then the function returns the zero-length string.

Example:

fn:substring-before("abcdefghi","d") returns "abc"

fn:substring-after

Returns a substring of a string that follows in a given string.

```
xs:string fn:substring-after($source_string as xs:string,
                             $bound_string   as xs:string)
```

Function opposite to fn:substring-before.

If the value of $source_string or $bound_string is the empty sequence, it is interpreted as the zero-length string.

If the value of $bound_string is the zero-length string, then the function returns the zero-length string.

If the value of $source_string does not contain a string that is equal to the value of $bound_string, then the function returns the zero-length string.

Example:

fn:substring-after("abcdefghi","d") returns "efghi"

String Functions that Use Pattern Matching

The three functions described in this section make use of a regular expression

syntax for pattern matching.

Caution: The regex used is the java regex see: http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html#sum

fn:matches

The function returns true if $input matches the regular expression otherwise, it returns false.

```
fn:matches($input  as xs:string?,
           $pattern  as xs:string)  as xs:boolean
```

If $input is the empty sequence, it is interpreted as the zero-length string.

The $pattern is a regular expression.

An error is raised if the value of $pattern is invalid.

Note: The variant of fn:matches accepting $flags parameter is not already supported.

```
fn:matches("abracadabra", "bra") returns false
fn:matches("abracadabra", ".*bra.*") returns true
fn:matches("abracadabra", "^a.*a$") returns true
fn:matches("abracadabra", "^bra") returns false
```

fn:replace

Returns the xs:string that is obtained by replacing each non-overlapping substring of $input that matches the given $pattern with an occurrence of the $replacement string.

```
fn:replace($input  as xs:string?,
           $pattern  as xs:string,
           $replacement  as xs:string) as xs:string
```

The $pattern is a regular expression.

If the $input is the empty sequence, it is interpreted as the zero-length string.

If the value of $bound_string is the zero-length string, then the function returns the zero-length string.

Note: The variant of fn:replace accepting $flags parameter is not already supported.

```
replace("abracadabra", "bra", "*") returns "a*cada*"
replace("abracadabra", "a.*a", "*") returns "*"
replace("abracadabra", "a.*?a", "*") returns "*c*bra"
replace("abracadabra", "a", "") returns "brcdbr"
replace("abracadabra", "a(.)", "a$1$1") returns "abbraccaddabbra"
replace("AAAA", "A+", "b") returns "b"
replace("AAAA", "A+?", "b") returns "bbbb"
replace("darted", "^(.*?)d(.*)$", "$1c$2") returns "carted". The first "d" is rep
```

fn:tokenize

Breaks the $input string into a sequence of strings, treating any substring that matches $pattern as a separator. The separators themselves are not returned.

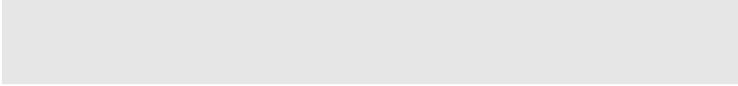
```
fn:tokenize($input as xs:string?,
            $pattern as xs:string) as xs:string+
```

The $pattern is a regular expression.

If the $input is the empty sequence, it is interpreted as the zero-length string.

Note: The variant of fn:tokenize accepting $flags parameter is not already supported.

```
fn:tokenize("1, 15, 24, 50", ",") returns ("1", "15", "24", "50")
```



Functions and Operators for anyURI

This section describes functions that take anyURI as arguments.

fn:resolve-uri

Caution: This function is not supported.

op:anyURI-equal

Caution: This function is not supported.

Functions and Operators on Boolean Values

This section describes functions and operators on the boolean datatype.

Additional Boolean Constructor Functions

The following additional constructor functions are defined on the boolean type.

fn:true

> **Caution:** fn:true() is not supported by the current implementation.

fn:false

> **Caution:** fn:false() is not supported by the current implementation.

Operators on Boolean Values

The following functions define the semantics of operators on boolean values.

op:boolean-equal

> **Caution:** op:boolean-equal() is not supported by the current implementation.

op:boolean-less-than

> **Caution:** op:boolean-less-than() is not supported by the current implementation.

op:boolean-greater-than

> **Caution:** op:boolean-greater-than() is not supported by the current implementation.

Functions on Boolean Values

The following functions are defined on boolean values:

fn:not

> **Caution:** fn:not() is not supported by the current implementation.

Functions and Operators on Durations, Dates and Times

This section discusses operations on the date and time types. It also discusses operations on two subtypes of xs:duration.

Comparisons of Duration, Date and Time Values

The following comparison operators are defined on the XML schema date, time and duration datatypes. Each operator takes two operands of the same type and returns a boolean result.

op:yearMonthDuration-equal

op:yearMonthDuration-less-than

op:yearMonthDuration-greater-than

op:dayTimeDuration-equal

op:dayTimeDuration-less-than

op:dayTimeDuration-greater-than

op:dateTime-equal

op:dateTime-less-than

op:dateTime-greater-than

op:date-equal

op:date-less-than

op:date-greater-than

op:time-equal

op:time-less-than

op:time-greater-than

op:gYearMonth-equal

op:gYear-equal

op:gMonthDay-equal

op:gMonth-equal

op:gDay-equal

Caution: These functions are not supported.

Component Extraction Functions on Durations, Dates and Times

The duration, date and time datatypes may be considered to be composite datatypes in that they contain distinct components. The extraction functions specified below extract a single component from a duration, date or time value. For xs:dateTime, xs:date and xs:time the localized value is used. To get the value of a component from the normalized value, the xs:dateTime, xs:date or xs:time must first be adjusted to UTC or timezone Z.

fn:years-from-duration

fn:months-from-duration

fn:days-from-duration

fn:hours-from-duration

fn:minutes-from-duration

fn:seconds-from-duration

fn:year-from-dateTime

fn:month-from-dateTime

fn:day-from-dateTime

- # fn:hours-from-dateTime
- # fn:minutes-from-dateTime
- # fn:seconds-from-dateTime
- # fn:timezone-from-dateTime
- # fn:year-from-date
- # fn:month-from-date
- # fn:day-from-date
- # fn:timezone-from-date
- # fn:hours-from-time
- # fn:minutes-from-time
- # fn:seconds-from-time
- # fn:timezone-from-time

Caution: These functions are not supported.

Arithmetic Functions on Durations

This section describes the Arithmetic Functions on Durations

- # op:add-yearMonthDurations
- # op:subtract-yearMonthDurations
- # op:multiply-yearMonthDuration
- # op:divide-yearMonthDuration
- # op:divide-yearMonthDuration-by-yearMonthDuration
- # op:add-dayTimeDurations
- # op:subtract-dayTimeDurations
- # op:multiply-dayTimeDuration
- # op:divide-dayTimeDuration
- # op:divide-dayTimeDuration-by-dayTimeDuration

Caution: These functions are not supported.

Timezone Adjustment on Dates and Time Values

These functions adjust an xs:dateTime, xs:date or xs:time value to the given or implicit timezone.

fn:adjust-dateTime-to-timezone

fn:adjust-date-to-timezone

fn:adjust-time-to-timezone

Caution: These functions are not supported.

Arithmetic Functions on Durations, Dates and Times

These functions support adding or subtracting a duration value to or from an xs:dateTime, an xs:date or an xs:time value.

If any of the arguments to the functions below is an xs:dateTime, xs:date or xs:time that does not contain an explicit timezone then, for the purpose of the operation, an implicit timezone, provided by the evaluation context, is assumed to be present as part of the value.

fn:subtract-dateTimes-yielding-yearMonthDuration

fn:subtract-dateTimes-yielding-dayTimeDuration

fn:subtract-dates-yielding-dayTimeDuration

fn:subtract-dates-yielding-yearMonthDuration

op:subtract-times

op:add-yearMonthDuration-to-dateTime

op:add-dayTimeDuration-to-dateTime

op:subtract-yearMonthDuration-from-dateTime

op:subtract-dayTimeDuration-from-dateTime

op:add-yearMonthDuration-to-date

op:add-dayTimeDuration-to-date

op:subtract-yearMonthDuration-from-date

op:subtract-dayTimeDuration-from-date

op:add-dayTimeDuration-to-time

op:subtract-dayTimeDuration-from-time

Caution: These functions are not supported.

Functions Related to QNames

Caution: Theses functions are not supported.

Functions and Operators on base64Binary and hexBinary

Caution: Theses functions are not supported.

Functions and Operators on NOTATION

Caution: Theses functions are not supported.

Functions and Operators on Nodes

This section describes functions and operators on nodes.

fn:name

Returns the name of a node, as a string that is either the zero-length string.

```
fn:name() as xs:string
fn:name($node as node()?) as xs:string
```

If the argument is omitted, it defaults to the context node. If there is no context node (that is, if the context item is not a node), the function returns the zero-length string. If the argument is supplied and is the empty sequence, the function returns the zero-length string. If the target node has no name, the function returns the zero-length string. Otherwise, the value returned is a string.

fn:local-name

Caution: This function is not supported.

fn:namespace-uri

> **Caution:** This function is not supported.

fn:number

> **Caution:** This function is not supported.

fn:lang

> **Caution:** This function is not supported.

op:is-same-node

> **Caution:** This function is not supported.

op:node-before

> **Caution:** This function is not supported.

op:node-after

> **Caution:** This function is not supported.

fn:root

> **Caution:** This function is not supported.

Functions and Operators on Sequences

This section describes the Functions and Operators on Sequences.

A sequence is an ordered collection of zero or more items. An item is either a node or an atomic value.

General Functions and Operators on Sequences

The following functions are defined on sequences.

fn:boolean

Caution: This function is not supported.

op:concatenate

Caution: This function is not supported.

fn:index-of

Caution: This function is not supported.

fn:empty

Caution: This function is not supported.

fn:exists

Caution: This function is not supported.

fn:distinct-values

Caution: This function is not supported.

fn:insert-before

Caution: This function is not supported.

fn:remove

Caution: This function is not supported.

fn:reverse

Caution: This function is not supported.

fn:subsequence

Caution: This function is not supported.

fn:unordered

> **Caution:** This function is not supported.

Functions That Test the Cardinality of Sequences

The following functions test the cardinality of their sequence arguments.

fn:zero-or-one

> **Caution:** This function is not supported.

fn:one-or-more

> **Caution:** This function is not supported.

fn:exactly-one

> **Caution:** This function is not supported.

Equals, Union, Intersection and Except

The following functions operates on Sequences.

fn:deep-equal

> **Caution:** This function is not supported.

op:union

> **Caution:** This function is not supported.

op:intersect

> **Caution:** This function is not supported.

op:except

> **Caution:** This function is not supported.

Aggregate Functions

Aggregate functions take a sequence as argument and return a single value computed from values in the sequence. Except for fn:count, the sequence must consist of values of a single type or one if its subtypes, or they must be numeric. xdt:untypedAtomic values are permitted in the input sequence and handled by special conversion rules. The type of the items in the sequence must also support certain operations.

fn:count

Returns the number of items in the sequence.

```
fn:count($arg as item()*) as xs:integer
```

The returned value is an integer with a value equal to the number of items in the sequence provided in parameter. Returns 0 if $arg is the empty sequence.

count() built-in function

```
(: Function call: the following query
    Loops on the tangent measurement data sets
    of a GOMOS file and counts the number of records
    it contains (i.e. it is not the most efficient
    way to compute the record numbers)
 :)

let $product := GOM_NL__2P/(schemas/gom/GOM_NL__2P.xml)GOM_NL__2P

for $tgtrecord in $product/tangent_mds

return
   <count>
      {fn:count($tgtrecord/tangent_mdsr)}
   </count>
```

fn:avg

Returns the average of a sequence of numbers.

```
fn:avg($arg as xdt:anyAtomicType*) as xdt:anyAtomicType?
```

The returned value has a type and a value depending on the type of the provided parameter:

If the node contains a value it is returned as average value. Otherwise an error is raised. The engine also controls the value is numeric. Otherwise an error is raised.

The engine controls the value is numeric and raise an error if not. The checked value is returned has average value.

The values of all the sequence items are extracted as described in the previous cases. All the values should then be summed and divided by the number of items in the sequence. In case of empty sequence an error is raised.

Caution: The returned type of this function is not restricted to a double as specified in the XQuery specifications but keeps as far as possible the types of the input data and convert them only to prevent losses (e.g. overflows, maximum precision reached, etc.).

avg() built-in function

```
fn:avg((3.0,4.0,5.0)) returns 4.0
```

fn:max

Returns the object with maximum value from a sequence of comparable items.

```
fn:max($arg as xdt:anyAtomicType) as xdt:anyAtomicType?
```

The returned value has a type and a value depending on the type of the provided parameter:

If the node contains a value it is returned as maximum value. Otherwise an error is raised.

The value is returned has maximum value.

The values of all the sequence items are extracted as described in the previous cases. All the values should then be compared to each others to extract their maximum. In case of empty sequence an error is raised.

Statistics

```
(: Statistical built-in functions :)

let $product := GOM_NL__2P/(schemas/gom/GOM_NL__2P.xml)GOM_NL__2P

for $tgtrecord in $product/tangent_mds

return
   <rec>
      <pos>
         fn:position()
      </pos>
      <min>
         fn:min($tgtrecord/tangent_mdsr/air)
      </min>
      <max>
         fn:max($tgtrecord/tangent_mdsr/air)
      </max>
      <avg>
         fn:avg($tgtrecord/tangent_mdsr/air)
      </avg>
   </rec>
```

Caution: The returned type of this function is not restricted to a double as specified in the XQuery specifications but keeps as far as possible the types of the input data and convert them only to prevent losses (e.g. overflows, maximum precision reached, etc.).

fn:min

Returns the object with minimum value from a sequence of comparable items.

```
fn:min($arg as xdt:anyAtomicType) as xdt:anyAtomicType?
```

The returned value has a type and a value depending on the type of the provided parameter:

If the node contains a value it is returned as minimum value. Otherwise an error is raised.

The value is returned has minimum value.

The values of all the sequence items are extracted as described in the previous cases. All the values should then be compared to each others to extract their minimum. In case of empty sequence an error is raised.

> **Caution:** The returned type of this function is not restricted to a double as specified in the XQuery specifications but keeps as far as possible the types of the input data and convert them only to prevent losses (e.g. overflows, maximum precision reached, etc.).

fn:sum

> **Caution:** This function is not supported.

Functions and Operators that Generate Sequences

The following functions generates Sequences.

op:to

> **Caution:** This function is not supported.

fn:id

> **Caution:** This function is not supported.

fn:idref

> **Caution:** This function is not supported.

fn:doc

Returns a node corresponding to the URI provided in parameter. sequence paramater.

```
fn:doc($uri as xs:string) as element()?
```

The input string shall be a valid URI or a empty string (i.e. ""). Otherwise, an error is raised. If the provided URI is a relative URI (i.e. opposite to absolute URI), the default base URI is used to resolve the absolute path.

It is important to notice that, if the URI does not match any node, the empty sequence is returned without error. Errors are only raised if the URI is not valid or if the underlying implementation encoutered an unrecoverable error. The function call may however produce a warning in the logger system, if, for example the protocol specifed by the URI is not supported.

fn:collection

Caution: This function is not supported.

Context Functions

The following functions are defined to obtain information from the evaluation context.

fn:position()

```
fn:position() as xs:integer
```

Use of position() built-in function

```
{-- position() built-in function --}

let $product := GOM_NL__2P/(schemas/gom/GOM_NL__2P.xml)GOM_NL__2P

for $tgtrecord in $product/tangent_mds/tangent_mdsr

return
   <rec>
      <pos>
         fn:position()
      </pos>
      <air>
         $tgtrecord/air
      </air>
   </rec>
```

fn:last()

```
fn:last() as xs:integer
```

If the context item is undefined, an error is raised.

The last item is completly defined in the section describing the concept of focus in the dynamic context. As a summary, the last item is the opposite item to the one designated by the fn:position function.

fn:current-dateTime()

```
fn:current-dateTime() as xs:dateTime
```

fn:current-date()

Caution: This function is not supported.

fn:current-time()

Caution: This function is not supported.

fn:default-collation()

Caution: This function is not supported.

fn:implicit-timezone

Caution: This function is not supported.

Chapter 14: Mathematical functions

A module of mathematical functions has been implemented in the DRB. The signature and arity of the functions defined are inherited from http://www.exslt.org/math. The functions are therefore defined in a namespace corresponding to the same URL. The DRB XQuery engine defines this namespace by default and bind it to the "math" prefix. The use of the prefix is mandatory otherwise altered or bounded to the default namespace.

math:abs()

The math:abs function returns the absolute value of a number. The output type of the function is the same as in input.

```
math:abs($value as xs:double) as xs:double
```

If the argument is not negative, the argument is returned.

If the argument is negative, the negation of the argument is returned.

- If the argument is positive zero or negative zero, the result is positive zero.
- If the argument is infinite, the result is positive infinity.
- If the argument is NaN, the result is NaN.

math:acos()

The math:acos function returns the arccosine value of a number.

```
math:acos($value as xs:double) as xs:double
```

The returned angle is expressed in radians and in the range of 0.0 through pi.

- If the argument is NaN or its absolute value is greater than 1, then the result is NaN.

math:asin()

The math:asin function returns the arcsine value of a number.

```
math:asin($value as xs:double) as xs:double
```

The returned angle is expressed in radians, in the range of -pi/2 through pi/2.

If the argument is NaN or its absolute value is greater than 1, then the result is NaN.

If the argument is zero, then the result is a zero with the same sign as the argument.

math:atan()

The math:atan function returns the arctangent value of an angle, in the range of -pi/2 through pi/2.

```
math:atan($value as xs:double) as xs:double
```

The returned angle is expressed in radians.

If the argument is NaN, then the result is NaN.

If the argument is zero, then the result is a zero with the same sign as the argument.

math:atan2()

The math:atan2 function converts rectangular coordinates (x, y) to polar (r, theta).

This method computes the phase theta by computing an arc tangent of y/x in the range of -pi to pi.

```
math:atan2($x as xs:double,
           $y as xs:double) as xs:double
```

The returned angle is expressed in radians.

If either argument is NaN, then the result is NaN.

If the first argument is positive zero and the second argument is positive, or the first argument is positive and finite and the second argument is positive infinity, then the result is positive zero.

If the first argument is negative zero and the second argument is positive, or the first argument is negative and finite and the second argument is positive infinity, then the result is negative zero.

If the first argument is positive zero and the second argument is negative, or the first argument is positive and finite and the second argument is negative infinity, then the result is the double value closest to pi.

If the first argument is negative zero and the second argument is negative, or the first argument is negative and finite and the second argument is negative infinity, then the result is the double value closest to -pi.

If the first argument is positive and the second argument is positive zero or negative zero, or the first argument is positive infinity and the second argument is finite, then the result is the double value closest to pi/2.

If the first argument is negative and the second argument is positive zero or negative zero, or the first argument is negative infinity and the second argument is finite, then the result is the double value closest to -pi/2.

If both arguments are positive infinity, then the result is the double value closest to pi/4.

If the first argument is positive infinity and the second argument is negative infinity, then the result is the double value closest to 3*pi/4.

If the first argument is negative infinity and the second argument is positive infinity, then the result is the double value closest to -pi/4.

If both arguments are negative infinity, then the result is the double value closest to -3*pi/4.

math:cos()

The math:cos function returns the trigonometric cosine of the passed angle argument.

```
math:cos($angle as xs:double) as xs:double
```

The $angle shall be expressed in radians.

If the argument is NaN or an infinity, then the result is NaN.

math:exp()

The math:exp function returns e (the base of natural logarithms) raised to a power. The return value is the power of the constant e. The constant e is Euler's constant, approximately equal to 2.178.

```
math:exp($value as xs:double) as xs:double
```

math:log()

The math:log function returns the natural logarithm of a double. The base is e.

```
math:log($value as xs:double) as xs:double
```

math:power()

The math:power function returns the value of a base expression taken to a specified power.

```
math:power($value as xs:double,
           $exponant as xs:double) as xs:double
```

math:random()

The math:random function returns a random double from 0 to 1.

```
math:random() as xs:double
```

math:sin()

The math:sin function returns sine of the passed argument.

```
double math:sin($angle as xs:double)
```

The $angle shall be expressed in radians.

If the argument is NaN or an infinity, then the result is NaN.

If the argument is zero, then the result is a zero with the same sign as the argument.

math:sqrt()

The math:sqrt function returns the square root of a double. If the argument is a negative number, the return value is zero.

```
math:sqrt($value as xs:double) as xs:double
```

math:stdev()

Returns the standard deviation of a sequence of numbers.

```
fn:stdev($arg as xdt:anyAtomicType*) as xdt:anyAtomicType?
```

The returned value has a type and a value depending on the type of the provided parameter:

Case of single node: If the node contains a value 0 is returned as standard deviation. Otherwise an error is raised. The engine also controls the value is numeric. Otherwise an error is raised.

Case of simple value: The engine controls the value is numeric and raise an error if not. 0 returned has standard deviation value.

Case of sequence of values or nodes: The values of all the

sequence items are extracted as described in the previous cases. All the values are then processed to compute the standard deviation of the set. In case of empty sequence an error is raised.

Caution: The returned type of this function is restricted to a double as specified in the XQuery specifications.

math:tan()

The math:tan function returns the tangent of the number passed as an argument.

```
math:tan($angle as xs:double) as xs:double
```

The $angle shall be expressed in radians.

If the argument is NaN or an infinity, then the result is NaN.

If the argument is zero, then the result is a zero with the same sign as the argument.

Chapter 15: DRB specific built-in functions

The DRB XQuery engine also provides a set of convenient functions. The interest of these functions has been identified during various projects nd experiences. They are however, not standard and are not classified according to their roles or themes. These functiosn shall be used with care since they may be modified among the engine releases as far as a best candidate or standard will be selected to host them.

drb:deep-invalid()

Returns the invalid nodes according to a schema. This function has two arguments and returns a collection of nodes that are invalid. The output nodes are wrapping the invalid nodes without keeping their hyerarchy. The nodes may be accessed to know their exact path, sibling or any other attribute. In addition to the attribute of the wrapped nodes, a set of validation specific attributes is appended. In particular these attributes are dedicated to the identification of the reason of invalidity.

```
deep-invalid($node as node, $schema as node?) as node*?
```

The first parameter $node specifies the node from which the validation shall start. The node and its children, if any, are recursivly checked. The $schema parameter may specify a schema to be used for the validation. This parameter is not mandatory if the specified node has already an associated schema. If the provided node has already an associated schema but a schema has also been provided through the $schema parameter, the new one overides the existing one. In case of invalid node, an attribute named "error" is attached to the node, containing the explicit error message.

evaluate-string()

Evaluates the string parameter as an XQuery script.

```
evaluate-string($script as xs:string) as item()*
```

The $script is evaluated considering the current dynamic context.

First, the script is parsed using the input static context part of the current dynamic context. The possible changes of context (e.g. new variables, change of default namespaces or new functions) will not alter the original context.

Secondly, the $script is evaluated considering a dynamic context derivated from the original context potentially altered during the static context.

The result of the dynamic evaluation constitutes the ouput of the functions.

> **Note:** This function is useful for evaluating scripts that are discovered while evaluating a script. As an example the dicovered script may be provided by a user through a graphical interface, or discovered from a configuration file.

evaluate-uri()

Evaluates the context of the file designated by the provided $uri as an XQuery script.

```
evaluate-uri($uri as xs:string) as item()*
```

The $uri is considered for retreiving and opening a file. The content of this file is considered as the input script that is evaluated with the current dynamic context.

An error is raised if $uri is not a valid URI or if the content of the designated file could not be read.

First, the script is parsed using the input static context part of the current dynamic context. The possible changes of context (e.g. new variables, change of default namespaces or new functions) will not alter the original context.

Secondly, the $script is evaluated considering a dynamic context derivated from the original context potentially altered during the static context.

The result of the dynamic evaluation constitutes the ouput of the functions.

> **Note:** This function is useful for evaluating scripts that are discovered while evaluating a script. As an example the dicovered script may be provided by a user through a graphical interface, or discovered from a configuration file.

index()

> **Caution:** index() function is deprecated !!! The use of fn:position() function shall be preferred to calls to this function.

Returns the index of the context item within the sequence of items currently being processed. The index is the one in the data source and not the index in the context (i.e. processed nodes). The index is an unsigned integer starting at 0. The index is useful for constraining joins to absolute occurrences in the data sources.

```
index() as xs:integer
```

The index() function applies only if the context item is a single node.

Caution: The index() function is not defined by the XQuery specifications.

parameter()

Caution: parameter() function is deprecated !!! The use of external variable declaration shall be preferred to the call of parameter() function. The external variable are resolved according to the external environment of evaluation of the quey script. Please refer to the user manual section describing the environment of evaluation for getting rules of external variable resolutions.

This function attempts to resolve variable parameters from the evaluation context. Actually in many cases, some values of the queries are not constant and may depend from the running environment may be with values to be provided by the user. Thus, this function enable to request a specific value to be at any moment in the query. As an example, many queries require an input document URI or a processing parameter.

```
parameter($param_name as xs:string,
          $type xs:string,
          $default as item()*) as item()*
```

drb:xml-string()

This function encodes a sequence of nodes in an output string formatted as a standard XML fragment with no prolog, possibly several root nodes separated with authorised XML characters (e.g. blank spaces, tabs, carriage returns etc.).

```
drb:xml-string($sequence as node()*) as xs:string
```

This function is useful for constructing presentation documents with embedded XML codes that should not be interpreted by the XQuery engine nor by the document procesor. The following example how this function may be used for displaying a fragment of XML Schema within a generated XHTML document. It could be part of a format specification or whatever technical document dedicated to users used to read XML framgents or examples:

The input XQuery

```
<html>
   <body>
   Here is the XML Schema:
   {
      drb:xml-string(doc("/my/schema.xsd")/schema/element[2])
   }
   </body>
</html>
```

Example of XHTML output

```
Here is the XML Schema:

<schema>
   <element name="value" type="xs:string"/>
</schema>
```

esa:utc-to-mjd()

This function converts a time epoch expressed in Universal Time Coordinated (UTC) to Modified Julian Days (MJD).

```
esa:utc-to-mjd($epoch as xs:dateTime, $dut1 as xs:double) as xs:double
```

$epoch is the input UTC time and $dut1 is the current delay between UTC and the Universal Time 1 (UT1).

The input XQuery

```
<date>
{
   (: This function returns 2451515.0000462965 days :)
   esa:utc-to-mjd(xs:dateTime("01-JAN-2000 12:00:00.000000"),
                  xs:double(2.0))
}
</date>
```

Chapter 16: XQuery conformance

This part will describe the different features of XQuery implemented, and the ones which are not supported yet.

Schemas and types

The use of xsl schemas and types are not supported yet. So It's impossible to type a variable, a function parameter or a node.

Namespace

The syntax for a namespace declaration or its use within a QName is supported. It is perfectly recognized for variables declaration and use. It is perfectly recognized for functions declaration and use (user-created functions). Problems occur with built-in functions and user-created functions with the same local name of a built-in function. Indeed only the local name is currently used to resolve such functions. Use of namespaces with nodes is syntactically correct but not interpreted.

Expression

Primary Expressions

Primary expressions are not totally supported. DoubleLiterals are not supported. Neither predefined entity reference.

Path Expressions

Path expressions are not totally supported. Kind tests, and unabbreviated syntax (child::, parent::) are not supported. Abbreviated syntax for axes is supported. The only axes supported are: child, parent(..), self(.), and attribute (@). A special syntax has been added, to allow casts of nodes within a path, using xschema. This syntax is not compliant at all with the specification, and has been added for personnal use.

Sequence Expressions

Sequence expressions are not totally supported. Filter expressions are not implemented. Combining node sequences (with UNION, INTERSECT, and EXCEPT) is supported, but results are not always those expected, due to the fact that equality between two nodes is not well implemented.

Arithmetic Expressions

IDIV Operator is not implemented. DIV Operator not always gives the relevant result (2 DIV 3 gives 1 instead of 1.5)

Comparison Expressions

Value comparisons and node comparisons are not implemented. General comparisons is supported but can give mistaken results in some cases implying nodes.

Logical Expressions

Logical Expressions are fully supported.

Constructors

Constructors are not fully supported. Direct element constructors are fully supported. Some other direct constructors are not supported, like comments () or processing instructions. Computed Constructors are not supported.

FLWOR Expressions

FLWOR Expressions are supported, except the Order By Clause and all keywords references this clause (like 'ascending' or 'empty least').

Conditionnal Expressions

Conditionnal expressions are fully supported.

Other Expressions

Ordered Expressions, Quantified Expressions, Expressions on SequenceTypes, Validate Expressions and Extension Expressions are not supported.

Modules

Version Declaration

The version declaration is supported except the keyword 'encoding'

Module Declaration

The module declaration is fully supported.

Base URI Declaration

The base URI declaration is supported but is not used.

Module Import Declaration

The Module Import Declaration is supported. Some problems are detected: -First if the graph of module imports contains a cycle, it is not detected. -Then module imports are currently transitives, while it should not be.

Variable Declaration

The Variable Declaration is supported.

Function Declaration

The Function Declaration is supported (except the issues described in the namespace and types section). A bug in the grammar prevent the user from declaring a function with no parameters.

Other Declarations

Boundary Space Declaration, Default Collation Declaration, Construction Declaration, Ordering Mode Declaration, Empty Order Declaration, Copy-Namespaces Declaration, Schema import and Option Declaration are not supported.

Part

2

DRB XML-Schema 1.0 processor

Chapter 1: Language overview

XML-Schema recommendation

The XML-Schema language has been developed by the World Wide Web Consortium (W3C). It is dedicated to the logical description of XML files and it major goal is to validated XML documents.

See also: The complete recommendation of the XML-Schema language can be downloaded from the W3C web page .

The XML-Schema language is compliant with XML 1.0 syntax and use XML-Namespaces to handle naming collision. A document contains a hierarchy of enclosed tags whose root is the schema itself.

Caution: Because XML-Schema has been designed for the valitation of XML documents, it does not describe the physical structure of the document. It is however opened enough to be extended with additional information.

The current implementation of the software is compatible with the XML-Schema recommendation but not compliant. This means all the items recognized by the software follows the W3C recommendation but others are missing. As a matter of fact the XML-Schema designed for this application can be read by any other compliant software. The following sections lists exhaustively the items currently supported.

A sample schema

Lets now consider a possible definition of a table. We want to specify that our table has a title and multiple lines. Each line in turn contains one or more cells embeding a float value. The corresponding schema sounds like this :

The Table Schema, table.xsd

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:annotation>
      <xsd:documentation>Example of table schema</xsd:documentation>
   </xsd:annotation>

   <!-- declaration of a table -->

   <xsd:element name="table" type="tableType"/>

   <!-- definition of a table content -->

   <xsd:complexType name="tableType">
      <xsd:attribute name="title" type="xsd:string" use="optional"/>
      <xsd:element name="line" type="lineType"
```

```
                    minOccurs="0" maxOccurs="unbounded"/>
      </xsd:complexType>

      <!-- definition of a line content -->
      <xsd:complexType name="lineType">
         <xsd:element name="cell" type="xsd:float"
                    minOccurs="1" maxOccurs="unbounded"/>
      </xsd:complexType>
   </xsd:schema>
```

We can see from this above example that the document use several xsd-prefixed tags, namely <xsd:schema>, <xsd:element>, <xsd:attribute>, <xsd:complexType>, <xsd:annotation> and <xsd:documentation>. These tags are parts of the XML-Schema language and will be detailed in the next section.

All schema document are enclosed by a <schema> and a <schema/>. These tags alert the parser that the document is really a schema. Inside this block we can put various data, mainly annotations, elements declaration and user-defined types. Futhermore the language allows external definitions to be placed anywhere providided they are in a different namespace.

The document starts with a global annotation intended for human reader. Our table schema declares only one element named table. It is followed by two complex types declaration so called 'tableType' and 'lineType'. The primer is a table content with an optional title attribute and an unbounded number of line elements. The second type is a line content with at least one cell of float value. The minOccurs parameter is set to 1 so that its occurence will not be null.

Chapter 2: XML-Schema basics

Schema declaration

The schema tag must be at the root of the document and should declare an URI (Unique Ressource Identifier) for the XML-Schema namespace. The shortcut prefix 'xs' has been choosen here by convention but any other one could be used instead. This namespace allows to avoid naming conflicts between schema's types and user's types.

The <schema> tag shall be declared as follow:

Syntax

```
<schema
  targetNamespace = xs:anyURI ?

  Content: (annotation?, include*,
            (complexType | element | attribute)*)
</schema>
```

targetNamespace - the namespace of the actual schema or empty. Every other document that wants to refer to this schema should use the URI declared in the targetNamespace.

Declaring a schema

```
<xsd:schema xmlns="http://foo.bar.org/xml-schemas/mySchema"
            xmlns:my="http://foo.bar.org/xml-schemas/mySchema"
            targetNamespace="http://foo.bar.org/xml-schemas/mySchema"
            xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Annotation declaration

Definition

We can put some annotations to all the component of the schema either xs:documentation for human and xs:appinfo for applications.

annotation

Content : (appinfo | documentation)*

appInfo

Content : ({any})*

documentation

Content : any textual information you want.

> **Note:** It is highly recommended to use the documentation markup as much as possible. Documented schemas are easier to understand. In addition the document markup is interpreted by the software that can display it on its graphical user interface.

Element declaration

xsd:element represents an element in the XML data model with a specified local name. It is associated to a type definition that restricts its content to either an atomic value (simple type) or named children and attributes (complex type).

Global elements are declared under xs:schema and can appear as root of a document. The namespace URI corresponds to the targetNamespace of the schema.

Declaring an element

```
<xsd:element name="sensing_start" type="xsd:dateTime"/>
```

Example:

<sensing_start>15-JUL-2000 06:30:00.000000</sensing_start>

Local elements are part of a complex type declaration and can appears as child of another element. The occurrence count may be constrained by a range (min/max).

The <element> tag shall be declared as follow:

Syntax

```
<element
  name      = xs:NCName
  ref       = xs:QName
  type      = xs:QName
  minOccurs = xs:nonNegativeInteger
  maxOccurs = xs:nonNegativeInteger | "unbounded"

  Content: (annotation?, (simpleType | complexType)?)
</element>
```

name - the local name of this element.

ref - the qualified name of referenced element.

type - the qualified name of a in-scope type declaration.

minOccurs - the minimum occurence of this element.

maxOccurs - the maximum occurence of this element. It may be "unbounded".

Note: Global elements shall have single occurrence

Attribute declaration

xsd:attribute represents an attribute in the XML data model with a specified local name. It is associated to a type definition that restricts its content to an atomic value. The XML-Schema provides common datatypes (decimal, string, dateTime) and allows to define new ones by restricting the value space.

Global attributes are declared under xs:schema and can be refered inside a complex type definition. The namespace URI corresponds to the targetNamespace of the schema.

The <attribute> tag shall be declared as follow:

Syntax

```
<attribute
  default = xs:string
  fixed   = xs:string
  name    = xs:NCName
  ref     = xs:QName
  type    = xs:QName
  use     = (optional | prohibited | required) : optional

  Content: (annotation?, simpleType?)
</attribute>
```

Declaring an attribute

```
<xsd:attribute name="unit" type="xsd:string"/>
```

default - The default value of this attribute or absent.

fixed - The fixed value of this attribute or absent.

name - The local name of the declared attribute.

ref - The qualified name of the refered attribute.

type - The qualified name of a simple type declaration. This attribute may be replaced by an anonymous simpleType definition.

use - The use of this declared attribute within a element. It is optional by default.

- optional - the attribute may be absent in the element instance
- prohibited - the attribute must not appear in the element instance
- required - the attribute must appear in the element instance

Modularization

It is also possible to include another schema with the <include> tag. Its schemaLocation attribute takes an URL pointing to an external schema. This is equivalent to a copy-paste operation.

Including a schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:include schemaLocation="http://foo.org/xml-schemas/mySchema.xsd"/>
   <!-- put here another definitions ... -->
</xsd:schema>
```

Syntax

Content : (annotation?)

- schemaLocation : the location of the schema to be included. This location is an absolute path or a relative path from the current directory.

Caution: It is highly recommended to use an absolute path only if the schema to be included is located in a reference repository. On the contrary a relative path is recommended if the included schema is closely linked the current one.

Chapter 3: Structure definition

Complex type declaration

A complex type is a compact representation of an element content in the XML data model. The syntax is similar to a grammar production rule in the language theory where the token corresponds to elements, attributes and atomic values.

Global complex types are declared under xs:schema and can be refered by their qualified name within an element, attribute or a type definition. The namespace URI corresponds to the targetNamespace of the schema.

Anonymous complex types are declared in the scope of an element or an attribute declaration.

Syntax

```
<complexType
  name = xs:NCName
  Content: (annotation?, ((all | choice | sequence), attribute*) |
                         (complexContent | simpleContent)?)
</complexContent>
```

name : the local name of this complex type.

Declaring a complex type

```
<xsd:complexType name="Address">
  <xsd:sequence>
    <xsd:element name="name"   type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city"   type="xsd:string"/>
      <xsd:element name="state"  type="xsd:string"/>
      <xsd:element name="zip"    type="xsd:integer"/>
    </xsd:sequence>
</xsd:complexType>
```

Model group definitions

The schema language has two connectors that can be applied to list of elements:

xs:sequence

xs:choice

Sequence model

The sequence connector fixes the order of the elements.

Syntax

```
<sequence
  Content: (annotation?, (choice | sequence | element)*)
</sequence>
```

Fixing elements order

```
<xsd:sequence>
  <xsd:element name="data_set_name"   type="xsd:string"/>
  <xsd:element name="ds_type" type="xsd:string"/>
  <xsd:element name="filename"   type="xsd:string"/>
  <xsd:element name="ds_offset"  type="xsd:long"/>
  <xsd:element name="total_ds_size"  type="xsd:long"/>
  <xsd:element name="number_dsr" type="xsd:int"/>
  <xsd:element name="dsr_length" type="xsd:int"/>
</xsd:sequence>
```

Attributes:

minOccurs : the minimum occurence of this sequence group in its parent.

maxOccurs : the maximum occurence of this sequence group in its parent.

Choice model

The choice connector selects one of the specified elements.

Syntax

```
<choice
  Content: (annotation?, (choice | sequence | element)*)
</choice>
```

Selecting one element in a group

```
<xsd:choice>
  <xsd:element name="OBSERVATION" type="RINEX_OBSERVATION"/>
  <xsd:element name="NAVIGATION"  type="RINEX_NAVIGATION"/>
</xsd:choice>
```

Attributes:

minOccurs : the minimum occurence of this choice group.

maxOccurs : the maximum occurence of this choice group.

Derivation methods

Extension

XML-Schema provides an object-oriented feature called extension that extend a complex type definition by appending new elements and/or attributes. Each derived type share the content of its base type. Therefore, the derived type is a valid instance of the base type.

The complexContent tag shall be declared inside a complexType declaration as follow:

Syntax

```
<complexContent
  Content: (annotation?, extension)
</complexContent>

<extension
  base = xs:QName
  Content: (annotation?, ((choice | sequence)?, attribute*))
</extension>
```

base - the qualified name of the base type.

Restriction

XML-Schema provides another derivation method called restriction that constrain a complex type definition by either reducing the range of an element occurrence, changing the use of some attributes or adding more facets to simple types. Therefore, the derived type is also a valid instance of the base type.

The complexContent tag shall be declared inside a complexType declaration as follow:

Syntax

```
<complexContent
  Content: (annotation?, restriction)
</complexContent>

<restriction
  base = xs:QName
  Content: (annotation?, ((choice | sequence)?, attribute*))
</restriction>
```

base - the qualified name of the base type.

Each declared particles shall matches without ambiguity the particles of the base types. The basic definition is simply a copy of the base type with altered occurrences.

Chapter 4: Datatype definition

Derivation

restriction

Simple content : (annotation?, minInclusive?, maxInclusive?, minExclusive?, maxExclusive?, minLength?, maxLength?, (pattern | enumeration)*

Attributes :

base : the name of the base type.

list

Content : (annotation?, minLength?, maxLength?, (pattern | enumeration)*

Attributes :

itemType: the name of the item type.

Facets

enumeration

Content : (annotation?)

Attributes :

value: The value of the enumeration facet.

Example

```
<xsd:element name="encoding">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="ASCII"/>
      <xsd:enumeration value="BINARY"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element name>
```

pattern

Content : (annotation?)

Attributes :

value: The value of the pattern facet.

Example

```
<xsd:element name="product_name">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="???_???_*.N1"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element name>
```

minLength

Content : (annotation?)

Attributes :

value: The value of the minLength facet.

maxLength

Content : (annotation?)

Attributes :

value: The value of the maxLength facet.

length

Content : (annotation?)

Attributes :

value: The value of the length facet.

whiteSpace

Caution: This facet is not supported.

fractionDigits

Caution: This facet is not supported.

totalDigits

Caution: This facet is not supported.

maxExclusive

Content : (annotation?)

Attributes :

value: The value of the maxExclusive facet.

maxInclusive

Content : (annotation?)

Attributes :

value: The value of the maxExclusive facet.

minExclusive

Content : (annotation?)

Attributes :

value: The value of the minExclusive facet.

minInclusive

Content : (annotation?)

Attributes :

value: The value of the minInclusive facet.

Example

```
<xsd:element name="price" type="priceType"/>

<xsd:simpleType name="priceType">
  <xsd:restriction base="xsd:decimal">
     <xsd:minInclusive value="0"/>
  </xsd:restriction>
</xsd:simpleType>
```

Chapter 5: Built-in datatypes

Decimal types

xs:decimal

xs:decimal represents a real number with arbitrary arithmetic precision.

```
("+" | "-") ? ("." [0-9]+) | ([0-9]+ "." [0-9]*)
```

The lexical representation consists of a sequence of integer digits and fraction digits separated by a period.

This notation allows leading and trailing zeros. The signs + and - indicate respectively positive and negative numbers.

Lexical forms

```
3.141592653589793238462643383279502884197169399375105820974944592
+129.500
-72
000542
0.123
```

The canonical representation removes the leading zeros, the trailing zeros and the positive sign.

Canonical forms

```
129.5
-72
542
.123
```

xs:decimal accepts the following facets:

- # totalDigits
- # fractionDigits
- # pattern
- # whiteSpace
- # enumeration
- # maxInclusive

- maxExclusive
- minInclusive
- minExclusive

xs:integer

xs:integer represents an arbitrary large integer number. It is derived from xs:decimal by setting the fractionDigits facet to 0.

The lexical representation consists of an optional sign followed by a sequence of digits.

This notation allows leading zeros but prohibits the fractional part.

Lexical forms

```
170141183460469231731687303715884105727
+129
-72
000542
```

The canonical representation removes the leading zeros and the positive sign.

Canonical forms

```
129
-72
542
```

xs:integer accepts the following facets:

- totalDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

xs:nonPositiveInteger

xs:nonPositiveInteger represents an integer number less than zero. It is derived from xs:integer by setting the maxInclusive facet to 0.

Lexical forms

```
-170141183460469231731687303715884105727
+0
000542
```

Canonical forms

```
0
-542
```

xs:nonPositiveInteger accepts the same facets that xs:integer.

xs:negativeInteger

xs:negativeInteger represents an integer number less than zero. It derives from xs:nonPositiveInteger by setting the maxInclusive facet to -1.

Lexical forms

```
-170141183460469231731687303715884105727
-000542
-1
```

The canonical representation removes the leading zeros.

Canonical forms

```
542
-1
```

xs:negativeInteger accepts the same facets that xs:integer.

xs:long

xs:long represents a 64-bits integer between -9223372036854775808 and 9223372036854775807.

Examples

```
-3000000000, 0, 123, 3000000000
```

xs:long accepts the same facets that xs:integer.

xs:int

xs:int represents a 32-bits integer between -2147483648 and 2147483647.

Examples

```
-1000000000, 0, 123, 45678
```

xs:int accepts the same facets that xs:integer.

xs:short

xs:short represents a 16-bits integer number between -32768 and 32767.

Examples

```
-10000, 9, 4253
```

xs:short accepts the same facets that xs:integer.

xs:byte

xs:byte represents an 8-bits integer between -128 and +127.

Examples

```
-3, 9, 112
```

xs:byte accepts the same facets that xs:integer.

xs:nonNegativeInteger

xs:nonNegativeInteger represents an integer number greater than zero. It derives from xs:integer by setting the minInclusive facet to 0.

Lexical forms

```
170141183460469231731687303715884105727
0
+000542
```

The canonical representation removes the leading zeros and the positive sign.

Canonical forms

```
129
72
```

xs:nonNegativeInteger accepts the same facets that xs:integer.

xs:unsignedLong

xs:long represents a 64-bits integer between 0 and 18446744073709551615.

Examples

```
0, 45678,  1000000000000000
```

xs:unsignedLong accepts the same facets that xs:integer.

xs:unsignedInt

xs:unsignedInt represents a 32-bits integer between 0 and 4294967295.

Examples

```
0, 123, 3000000000
```

xs:unsignedInt accepts the same facets that xs:integer.

xs:unsignedShort

xs:unsignedShort represents a 16-bits integer between 0 and 65535.

Examples

```
0, 123, 40000
```

xs:unsignedShort accepts the same facets that xs:integer.

xs:unsignedByte

xs:unsignedByte represents an 8-bits integer between 0 and 255.

Examples

```
3, 9, 112
```

xs:unsignedByte accepts the same facets that xs:integer.

xs:float

xs:float represents a 32-bits floating number as defined by the IEEE 754-1985.

The lexical representation consists of a sequence of integer digits and fraction digits separated by a period. Scientific notation with an optional exponent "E" is allowed.

Examples

```
-1.57e-15
+129
3.1415
-Infinity
Infinity
NaN
```

xs:float accepts the following facets:

- # pattern
- # enumeration
- # whiteSpace
- # maxInclusive
- # maxExclusive
- # minInclusive
- # minExclusive

xs:double

xs:double represents a 64-bits floating number as defined by the IEEE 754-1985.

The lexical representation consists of a sequence of integer digits and fraction digits separated by a period. Scientific notation with an optional exponent "E" is allowed.

Examples

```
-1.57e-200
NaN
-Infinity
Infinity
```

xs:double accepts the following facets:

pattern

enumeration

whiteSpace

maxInclusive

maxExclusive

minInclusive

minExclusive

xs:boolean

xs:boolean represents a single bit value.

Lexical forms

```
true
false
1
0
```

Canonical forms

```
true
false
```

xs:boolean accepts the following facets:

pattern

whiteSpace

Date, Time and Duration types

xs:dateTime

xs:dateTime represents a date with the following sub-components:

year as a fixed integer
month as three characters
day as an integer
hour as an integer
minute as an integer
second as a fixed decimal

Caution: Conversly to the XML-Schema recommendation, the construction of an xs:dateTime from a string shall not conform to the ISO-8601 but the ASCII form of European Space Agency (ESA) ENVISAT MJD date defined in the MGA-GS-2009 ENVISAT PDS Format Specification, Annex. An explicit example of this format may be "01-JAN-2005 00:00:00.000000". All characters shall always be expressed.

Lexical representation

```
DD-MMM-YYYY hh:mm:ss.ssssss

DD   = a two-digit numeral that represents the days from 01 to 31.
MMM  = three characters that represents the abbreviated month.
YYYY = a four-digit numeral that represents the yeas from 0000 to 9999.
hh   = a two-digit numeral that represents the hour from 0 to 23.
mm   = a two-digit numeral that represents the minutes from 0 to 59.
ss   = a 2.6-digit numeral that represents the seconds with a fractional part.
```

Example:

01-JAN-2005 00:00:00.000000

xs:dateTime accepts the following facets:

length
minLength
maxLength
pattern
enumeration
whiteSpace

xs:duration

xs:duration represents a duration of time with the following sub-components:

year as an integer
month as an integer
day as an integer
hour as an integer
minute as an integer
second as a decimal

Lexical representation

```
[-]? "P" (Years "Y")? (Months "M")? (Days "D")?
    ("T" (Hours "H")? (Minutes "M")? (Seconds "S")? )?
```

The designator "P" must always be present. The "T" symbol separates the date and time components of the duration. The designator "T" must be absent if and only if all of the time items are absent.

Almost all couples of value-unit (e.g. 2M or 10H) are optional but at least one shall be provided.

Example

```
P1Y2M3DT10H30M  = 1 year, 2 months, 3 days, 10 hours and 30 minutes
P10Y            = 10 years
PT20.86S        = 20.86 seconds
```

xs:duration has the following constraining facets:

pattern
enumeration
whiteSpace
maxInclusive
maxExclusive
minInclusive
minExclusive

String literals

xs:string

xs:string represents character strings as defined by XML 1.0 (Second Edition).

```
"Hello World!"
```

xs:string accepts the following facets:

- # length
- # minLength
- # maxLength
- # pattern
- # enumeration
- # whiteSpace

xs:normalizedString

xs:normalizedString represents character strings with normalized blanks. Such strings never contains the carriage return nor tab characters.

xs:normalizedString accepts the same facets than xs:string.

xs:token

xs:token represents tokenized strings. Such strings never contains the carriage return, line feed nor tab characters, leading or trailing spaces, internal sequences of two or more spaces.

xs:token accepts the same facets than xs:string.

xs:language

xs:language represents a natural language identifier as defined by the IETF (RFC 1766).

The lexical representation consists of two letters that identify a country.

Examples

```
en     English
fr     French
it     Italian
```

xs:language accepts the same facets than xs:string.

xs:Name

xs:Name represents a XML standard name. Such strings must contains letter, digits and basic separators. This type shall never starts with a digit.

Lexical representation

```
Name ::= (Letter | '_' | ':') (NameChar)*
NameChar ::= Letter | Digit | '.' | '-' | '_' | ':'
   |  CombiningChar | Extender
```

xs:Name accepts the same facets than xs:string.

xs:NCName

xs:NCName represents a non-colonized name. Such strings must contains letter, digits and basic separators. The lexical representation correspond to xs:Name without the colon character ':'.

xs:NCName accepts the same facets than xs:string.

xs:anyURI

xs:anyURI represents a Uniform Resource Identifier (URI).

Examples

```
http://www.gael.fr
http://www.w3.org
```

xs:anyURI accepts the same facets than xs:string.

XML 1.0 datatypes

The XML-Schema language retains the XML 1.0 types for compatibility with the XML 1.0 DTDs. They shall be used only within attribute declarations.

xs:NOTATION

Caution: This type is not supported.

xs:NMTOKEN

xs:NMTOKEN represents a string without blanks or special character. It matches the production rule:

Lexical representation

```
NMTOKEN ::= (Letter | Digit | '-' | '.')+
```

xs:NMTOKENS

xs:NMTOKENS represents a list of token.

The lexical representation consists of a list of xs:NMTOKEN separated by a blank character.

xs:ID

xs:ID represents a unique identifier. It is derived from the xs:NCName type.

xs:ID accepts the same facets than xs:string.

xs:IDREF

xs:IDREF represents an identifier to an ID. It is derived from the xs:NCName type.

xs:IDREF accepts the same facets than xs:string.

xs:IDREFS

xs:IDREFS represents a list of xs:NCName separated by a blank character. Each item shall refers to an identifier.

xs:ENTITY

xs:ENTITY represents an entity available in the environnement.

The lexical representation correponds to non-colonized name as specified by the xs:NCName type.

xs:ENTITIES

xs:ENTITIES represents a list of entity.

The lexical representation consists of a list of xs:ENTITY separated by a blank character.

Part 3

Structured Data Format

Chapter 1: General overview

Introduction

The Structured Data File (SDF) is an implementation of DRB that handles data sources (file, stream, string literals) with a user-defined format.

Definition

A data file is usually defined as a set of adjacent blocks with various lengths. A Structured Data File is a file containing ASCII or BINARY data which can be described as a hierarchy of data blocks. These blocks are organised in a "tree model" as a file system does with directories, sub-directories and files.

In addition to the low-level description of the data this representation allows to define higher level blocks to create logical groups. And therefore permits to browse the data in a non-linear way.

Note: In a tree representation, the low-level blocks are called "leaves" and high-level blocks are called "nodes". The top-level node is the "root node" and nodes are commonly named "parent" and "child".

Data description

A logical description is first needed to set up the structure of the tree. To do so the XML-Schema language has been choosen. It allows a large set of simple and complex types to build the nodes and leaves hierarchy.

See also: The complete recommendation of the XML-Schema language can be downloaded from the W3C web page .

Once the logical view available, a physical description is mandatory to access the data themselves. XML-Schema language has been designed for logical structure validation but using specific annotations it is possible to extend its functionalities.

XML-Schema sample with SDF annotations

The sample below has been extracted from the "ENVISAT.xsd" schema. This is the description of the "data set name" element in the "data set descriptor".

Element sample from ENVISAT.xsd

```
<xsd:element name="ds_name" type="xsd:string">
   <xsd:annotation>
      <xsd:documentation>
         Data Set Name - Name describing the data set.
         Characters not used are blanked.
      </xsd:documentation>
      <xsd:appinfo>
         <sdf:block>
            <sdf:padding type="header">9</sdf:padding>
            <sdf:padding type="footer">2</sdf:padding>
            <sdf:encoding>ASCII</sdf:encoding>
            <sdf:length>28</sdf:length>
         </sdf:block>
      </xsd:appinfo>
   </xsd:annotation>
</xsd:element>
```

Generic Schema syntax is described in previous chapter. Additional SDF tags are described in the next section.

Chapter 2: Adding physical description

Namespace consideration

SDF has been designed to use XML-Schema and create additional namespaces in order to describe the data from both the logical and the physical parts. Additional namespace "sdf", must be defined as below :

Declaring SDF namespace

```
<xsd:schema xmlns:sdf="http://www.gael.fr/2005/04/drb/sdf"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   ...
</xsd:schema>
```

The block description

Definition

For all elements and types a block description node can be added to the children nodes list (such as "annotation"). This is the mandatory information for the physical description. Various contents of the block are described below. The sdf:block is declared within an xsd:appinfo node as follow:

Syntax

```
<sdf:block
  query = xs:string : "."
  Content: occurrence?,
           length?,
           offset?,
           encoding?,
           padding?,
           delimiter?,
           array?,
           signature?,
           numericFormat?
</sdf:block>
```

query - This is an XQuery script that specifies the file containing the block data. It corresponds to the base node by default.

Single file format

sdf:block applies to:

- xsd:element

xsd:simpleType

xsd:complexType

xsd:choice

Block description for an element

```
<xsd:element name="ds_offset" type="xsd:long">
   <xsd:annotation>
      <xsd:documentation>
         DS Offset in bytes Gives the position of the first byte of the
         corresponding DS with respect to the whole product.
      </xsd:documentation>
      <xsd:appinfo>
        <sdf:block>
          <sdf:padding type="header">10</sdf:padding>
          <sdf:padding type="footer">8</sdf:padding>
          <sdf:encoding>ASCII</sdf:encoding>
          <sdf:length>21</sdf:length>
       </sdf:block>
      </xsd:appinfo>
   </xsd:annotation>
</xsd:element>
```

Block description for a complexType

```
<xsd:complexType name="DSD">
   <xsd:annotation>
      ...
      <xsd:appinfo>
         <sdf:block>
            ...
         </sdf:block>
      </xsd:appinfo>
   </xsd:annotation>
   <xsd:element ...>
      ...
   </xsd:element>
   <xsd:element ...>
      ...
   </xsd:element>
</xsd:complexType>
```

Note: "annotation" node is important because it is usually composed of a "documentation" child. This documentation can be used over several applications such as tooltip display in Derby, online help or paper documentation generation, etc.

Multiple files format

This is a new feature of SDF 2.2 that allow formats composed of several files. In that case, the base node is typically a directory containing at least two files. Thanks to the query attribute, SDF can open a file at run-time.

For example, a Landsat imagery product has eight files identified by their suffix name.

B10 - the visible band 1 (VNIR)

B20 - the visible band 2 (VNIR)

B30 - the visible band 3 (VNIR)

B40 - the infrared band 4(SWIR)

B50 - the infrared band 5 (SWIR)

B61 - the thermal band 1(THM)

B62 - the thermal band 2 (THM)

B80 - the panchromatic band 8 (PAN)

Example of multi-file format : Landsat FAST

```
<xsd:element name="FAST" type="FAST"/>

<xsd:complexType name="FAST">
  <xsd:sequence>
    <xsd:element name="vnir_band" type="Band">
      <xsd:annotation>
        <xsd:appinfo>
          <sdf:block query="./*[fn:matches(name(), 'L7.*_B10.FST')]"/>
        </xsd:appinfo>
      </xsd:annotation>
    </xsd:element>

    <xsd:element name="vnir_band" type="Band">
      <xsd:annotation>
        <xsd:appinfo>
          <sdf:block query="./*[fn:matches(name(), 'L7.*_B20.FST')]"/>
        </xsd:appinfo>
      </xsd:annotation>
    </xsd:element>

    ...

    <xsd:element name="pan_band" type="Band">
      <xsd:annotation>
        <xsd:appinfo>
          <sdf:block query="./*[fn:matches(name(), 'L7.*_B40.FST')]"/>
        </xsd:appinfo>
      </xsd:annotation>
    </xsd:element>
    ...
  </xsd:sequence>
</xsd:complexType>
```

The array definition

When the datatype of an element is xs:list, it is possible to set the attributes relying to array item by using an sdf:array node. The supported children are the following

:

occurrence: define the size of the array.

length: define the length of each item in the array.

delimiter: define the separator between each item in the array.

Length and occurrence of an array item

```
<xsd:element name="toa_rad">
  <xsd:annotation>
    <xsd:appinfo>
      <sdf:block>
         <sdf:array>
            <sdf:occurrence>1012</sdf:occurrence>
            <sdf:length>2</sdf:length>
         </Sdf:array>
      </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:simpleType>
     <xsd:list itemType="xs:unsignedShort"/>
  </xsd:simpleType>
</xsd:element>
```

Occurrence count definition

Definition

Purpose of this node is to set the occurrence count of the element in the Structured Data File. This descriptor is not mandatory and the default value is set to 1 if omitted. The Occurrence count of an element can be fixed for all datafile using this schema or can depend on the data file instance. Therefore you can use this descriptor in two different ways : By setting a fixed value or setting a query to process with the file instance.

Syntax

```
<sdf:occurrence
  query = xs:string ?
  constant = xs:boolean : true

  simpleContent = xs:nonNegativeInteger | unbounded
</sdf:occurrence>
```

query - An XQuery script that computes the occurrence value. This is usually an XPath but complex expressions are allowed. The result shall be an item castable as a long integer (64 bits). In case of empty result, the occurrence is set to zero.

constant - When a query is present, this attribute indicates wether the occurrence shall be considered constant. This assumption is true by default.

Fixed occurrence

Since version 2.2, SDF can compute the occurrence at run-time depending on the minOccurs and maxOccurs attributes.

One occurrence by default

```
<xsd:element name="tie_pt_ads" type="MER_RR__1P_ADSR_tie_pt"/>
```

Default values are usefull to reduce schema size and maintenance. In the above example, no occurrence node has been defined, therefore 1 occurrence of the element is considered by default.

Fixed Occurrence count

```
<xsd:element name="tie_pt_ads" type="MER_RR__1P_ADSR_tie_pt">
  <xsd:annotation>
    <xsd:appinfo>
       <sdf:block>
          <sdf:occurrence>38</sdf:occurrence>
       </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

In this example, all data files opened with this schema will have 38 occurrences of the "tie_pt_ads" structure.

Dynamic occurrence

In case of query the "sdf:occurrence" node doesn't need any value. The query definition is the string value of the "query" attribute. Therefore the node can be closed directly ("<sdf:occurrence query="..."/>") instead of using "<sdf:occurrence ...></sdf:occurrence>".

Occurrence count by Query

```
<xsd:element name="tie_pt_ads" type="MER_RR__1P_ADSR_tie_pt">
  <xsd:annotation>
    <xsd:appinfo>
       <sdf:block>
          <sdf:occurrence query="../sph/tie_pt_ads/num_dsr"/>
       </sdf:block>
    </xsd:appinfo>
```

```
    </xsd:annotation>
  </xsd:element>
```

Unknown occurrence

When the occurrence cannot be defined either by a fixed value or with a query, the SDF engine tries fo fill the available space. It will increase the occurrence count until the end of the file or the bounds of the parent block.

Occurrence limited by the file size

```
  <xsd:element name="record" type="RINEX_OBSERVATION_record"
               minOccurs="0" maxOccurs="unbounded">
    <xsd:annotation>
      <xsd:appinfo>
         <sdf:block>
            <sdf:occurrence>unbounded</sdf:occurrence>
         </sdf:block>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:element>
```

Array size

Within an sdf:array, this tag defines the number of item in an array.

Fixing the number of item in an array

```
<xsd:element name="geocentric_coords" type="coord_list">
  <xsd:annotation>
    <xsd:appinfo>
      <sdf:block>
        <sdf:array>
          <sdf:occurrence>3</sdf:occurrence>
        </sdf:array>
      </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

Arrays have also default size based on the xsd:length facet. The previous example could be re-written as follow:

Using default array size

```
<xsd:simpleType name="coord_list">
  <xsd:list itemType="xsd:decimal">
    <xsd:length value="3"/>
  </xsd:list>
</xsd:simpleType>
```

```
<xsd:element name="geocentric_coords" type="coord_list"/>
```

Length definition

Definition

This node is used to set the length of an element. It is mandatory for the "leaves" elements because it sets the number of bytes to be read to access the data in the file. For complex elements and types, it is not recommended to set the length (unless you want to force the value) because it will result from the sum of all child lengths, paddings, etc. As the occurrence count the length can be set to a fixed value or using query.

Syntax

```
<sdf:length
  query = xs:string ?
  constant = xs:boolean : true
  unit = byte | bit : byte

  simpleContent = xs:nonNegativeInteger | unbounded
</sdf:length>
```

query - An XQuery script that computes the length value. This is usually an XPath but complex expressions are allowed. The result shall be an item castable as a unsigned long integer (64 bits). In case of empty result, the length is set to zero.

constant - When a query is present, this attribute indicates wether the block length shall be considered constant. This assumption is true by default.

unit - The unit of the length value.

Fixed length

Fixed byte length

```
    <xsd:element name="radiance" type="xsd:integer">
       <xsd:annotation>
          <xsd:appinfo>
             <sdf:block>
                <sdf:encoding>BINARY</sdf:encoding>
                <sdf:length>12</sdf:length>
             </sdf:block>
          </xsd:appinfo>
       </xsd:annotation>
    </xsd:element>
```

This schema defines an binary integer encoded with 12 bytes.

Dynamic length

In case of length defined by a query, we considered by default that the length is constant e.g. the query does not depends on the current occurrence. If the value of the sdf:constant attribute is false then SDF will compute the query each time.

> **Note:** If the length is defined by a query, no value is necessary for the node, which is directly closed ("<sdf:length .../>")

Byte length using query

```
<xsd:element name="mds" type="MDS"
   <xsd:annotation>
      <xsd:documentation>
         Measurement data set
      </xsd:documentation>
      <xsd:appinfo>
         <sdf:block>
            <sdf:length query="../sph/dsd/ds_size"/>
         </sdf:block>
      </xsd:appinfo>
   </xsd:annotation>
</xsd:element>
```

Units

Length value can be defined in bytes or in bits. A "unit" attribute can be added to set "byte" or "bit" unit. If the "unit" attribute is ommited, the "byte" unit is set by default.

Bit length

```
<xsd:element name="seq_count" type="xsd:unsignedShort">
   <xsd:annotation>
      <xsd:documentation>
         Source Segmentation Counter
      </xsd:documentation>
      <xsd:appinfo>
         <sdf:block>
            <sdf:length unit="bit">14</sdf:length>
         </sdf:block>
      </xsd:appinfo>
   </xsd:annotation>
</xsd:element>
```

Unspecified length

The length is not mandatory for element with complex content since it can be computed by summing the length of the children elements.

In case of binary data, a default length is used.

Binary length of schema datatypes

```
xsd:unsignedByte     1 byte
xsd:byte             1 byte
xsd:short            2 bytes
xsd:unsignedShort    2 bytes
xsd:int              4 bytes
xsd:unsignedInt      4 bytes
xsd:long             8 bytes
xsd:unsignedLong     8 bytes
xsd:long             8 bytes
xsd:unsignedLong     8 bytes

xsd:float            4 bytes
xsd:double           8 bytes

xsd:dateTime        12 bytes
```

In case of text data (ASCII), SDF will parse the files by using the lexical representation of the datatype.

Variable byte length

```
<xsd:element name="ds_offset" type="xsd:integer">
  <xsd:annotation>
    <xsd:documentation>
      Offset of the dataset.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:appinfo>
    <sdf:block>
      <sdf:encoding>ASCII</sdf:encoding>
    </sdf:block>
  </xsd:appinfo>
</xsd:element>
```

Note: Datatype that have not a default length nor a pattern must declare an sdf:lenght with value "unbounded".

Array item length

Within an sdf:array, this tag defines the length of an array item.

Fixing the item length in an array

```
<xsd:element name="geocentric_coords" type="coord_list">
```

```
    <xsd:annotation>
      <xsd:appinfo>
        <sdf:block>
          <sdf:array>
            <sdf:length>3</sdf:length>
          </sdf:array>
        </sdf:block>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:element>
```

Offset definition

Definition

Purpose of this node is to set the offset of an element. By default, the offset value is considered as a relative offset from the previous sibling element. This means a shift between the last byte of the last occurrence of the previous element and the first byte of the current one. In most cases, an element is following its previous sibling without any gap therefore no relative offset is needed. If the "sdf:offset" node is omitted, the value is set to 0 by default.

> **Note:** In order to read the data at the right offset in the file, the absolute offset is computed from previous element structures such as occurrence counts, lengths and relative offsets.

Syntax

```
<sdf:offset
  query = xs:string ?
  unit = byte | bit : byte
  origin = previous | parent | root : previous

  simpleContent = xs:integer : 0
</sdf:offset>
```

query - An XQuery script that computes the offset value. This is usually an XPath but complex expressions are allowed. The result shall be a single item castable as a unsigned long integer (64 bits). In case of empty result, the offset is set to zero.

unit - The unit of the offset value.

origin - The origin of this offset for relative computation.

Units

Offset value can be defined in bytes or in bits. A "unit" attribute can be added to set "byte" or "bit" unit. If the "unit" attribute is ommited, the "byte" unit is set by

default.

Origin

SDF can compute the absolute position in the file from a user-defined origin:

- previous - the offset is relative to the previous-sibling node.
- parent - the offset is relative to the parent node.
- root -the offset is absolute (e.g. from the beginning of the file)

Fixed offset

In the example below we define a gap of 41 bytes between the time element and its previous sibling (the third occurrence of point element).

Offset definition

```
<xsd:element name="point" type="POINT"
             minOccurs="3" maxOccurs="3"/>

<xsd:element name="time" type="xsd:dateTime">
  <xsd:annotation>
    <xsd:appinfo>
      <sdf:block>
        <sdf:offset>41</sdf:offset>
        <sdf:length>12</sdf:length>
      </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

Dynamic offset

If the length is defined by a query, no value is necessary for the node, which is directly closed ("<sdf:offset .../>")

In this example, we define an absolute offset retreived in the DSD node thanks to an XPath.

Offset definition with query and origin

```
<xsd:element name="flag_mds" type="MER_RR__1P_flags">
  <xsd:annotation>
    <xsd:appinfo>
      <sdf:block>
        <sdf:offset origin="root" query="../sph/flag_dsd/ds_offset"/>
      </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

Padding definition

This node is used to define a gap (offset) inside an element. Two types are available : the "header" padding and the "footer" padding. They are respectively the offset to skip before and after the element value to be extracted. They can be set together or only one of them. If the "sdf:padding" node is omitted the value is set to 0 by default.

```
<sdf:padding
   type = (header | footer)
   content = xs:string ?>
   simpleContent: (xs:nonNegtiveInteger?)
</sdf:padding>
```

Header and footer padding definition

```
<xsd:element name="proc_time" type="xsd:dateTime">
  <xsd:annotation>
    <xsd:appinfo>
       <sdf:block>
          <sdf:padding type="header">11</sdf:padding>
          <sdf:padding type="footer">2</sdf:padding>
          <sdf:length>27</sdf:length>
          <sdf:encoding>ASCII</sdf:encoding>
       </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

Example:

PROC_TIME="10-SEP-2004 05:45:17.766357"\n

In the above example a ASCII time is described using the keyword="value" format. To extract the UTC time value it is needed to skip the keyword proc_time=" (11 bytes) and the ending "\n (2 bytes).

Chapter 3: Formatting atomic values

Encoding definition

In a logical representation the encoding is not needed but in a physical representation you need to know how to extract the data from the file. Therefore this node is used to set the encoding method of the element. The possible values are ASCII or BINARY. If the "sdf:encoding" node is omitted the encoding is inherited from the parent. The root has a BINARY encoding by default.

Syntax

```
<sdf:encoding
   simpleContent: "BINARY" | "ASCII" as xs:string
</sdf:encoding>
```

ASCII encoding string element

```
<xsd:element name="proc_stage" type="xsd:string">
  <xsd:annotation>
    <xsd:appinfo>
       <sdf:block>
          <sdf:encoding>ASCII</sdf:encoding>
          <sdf:length>1</sdf:length>
       </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

ASCII encoding for time element

```
<xsd:element name="proc_time" type="xsd:dateTime">
  <xsd:annotation>
    <xsd:appinfo>
       <sdf:block>
          <sdf:length>27</sdf:length>
          <sdf:encoding>ASCII</sdf:encoding>
       </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

Binary (by default) encoding for time element

```
<xsd:element name="time" type="xsd:dateTime">
  <xsd:annotation>
    <xsd:appinfo>
       <sdf:block>
```

```
            <sdf:length>12</sdf:length>
          </sdf:block>
        </xsd:appinfo>
      </xsd:annotation>
    </xsd:element>
```

In the first of the above examples, a size 1 string element (a character) is set with the ASCII encoding. In the second and third examples we can see the difference between a ASCII and a BINARY representation of a time. In case of ASCII encoding the time is extracted as UTC in a 27 characters string. In case of binary encoding the time is extracted as a MJD, using 12 bytes.

> **Note:** Encoding method is also important to get values from numeric elements. In case of float a BINARY encoding means the IEEE representation is used to extract the value.

Endianness

SDF handles two binary encoding for integers.

- # Most Significant Byte first (MSB)
- # Less Significant Byte first (LSB)

Syntax

```
<sdf:byteOrder
  simpleContent: MSB | LSB : default = MSB
</sdf:byteOrder>
```

Most Significant Byte (MSB) ordering

The Most Significant Byte encoding (MSB) also known as big-endian writes the bytes from left to right.

For example, the hexadecimal number 0x1ED8 (e.g. 7896) encoded as a short (2-bytes integer) is written 1E D8.

Encoding a short in MSB

```
<xsd:element name="radiance" type="xsd:short">
  <xsd:annotation>
    <xsd:appinfo>
      <sdf:block>
        <sdf:encoding>BINARY</sdf:encoding>
        <sdf:length>2</sdf:length>
        <sdf:byteOrder>MSB</sdf:byteOrder>
      </sdf:block>
    </xsd:appinfo>
```

```
  </xsd:annotation>
</xsd:element>
```

Less Significant Byte (LSB) ordering

The Less Significant Byte encoding (LSB) also known as little-endian writes the bytes from right to left.

For example, the hexadecimal number 0x03E03F38 (e.g. 65027896) encoded as an int (4-bytes integer) is written in reverse order 38 3F E0 03.

Encoding an int in LSB

```
<xsd:element name="tie_point" type="xsd:int">
  <xsd:annotation>
    <xsd:appinfo>
      <sdf:block>
        <sdf:encoding>BINARY</sdf:encoding>
        <sdf:length>4</sdf:length>
        <sdf:byteOrder>LSB</sdf:byteOrder>
      </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

Delimiter definition

This node define the separator character between two or more elements. The default delimiter is a space character for ASCII encoding and empty for BINARY encoding.

Adding a comma between elements

```
<xsd:element name="record" type="xsd:double" minOccurs="4">
  <xsd:annotation>
    <xsd:appinfo>
       <sdf:block>
          <sdf:delimiter>,</sdf:delimiter>
       </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

Example:

1.4901E-08, 2.2352E-08, -5.9605E-08, -1.1921E-07

Hereunder, an alternative schema based on the xs:list type.

Array with comma-separated values

```
<xsd:element name="csv">
  <xsd:annotation>
    <xsd:appinfo>
      <sdf:block>
         <sdf:array>
            <sdf:occurrence>15</sdf:occurrence>
            <sdf:delimiter>,</sdf:delimiter>
         </Sdf:array>
      </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:simpleType>
     <xsd:list itemType="xs:integer"/>
  </xsd:simpleType>
</xsd:element>
```

Numeric format

Purpose of the sdf:numericFormat node is to set the lexical representation of a numeric value. It controls the following features:

integer digits

fraction digits

exponent digits

prefix

suffix

The format is represented by a pattern that matches the production rules:

Pattern syntax

```
Pattern ::=
        Prefix? Number Suffix?
Prefix :::=
        any Unicode characters except \uFFFE, \uFFFF, and special characters
Suffix ::=
        any Unicode characters except \uFFFE, \uFFFF, and special characters
Number ::=
        Integer Exponent?
   |    Integer '.' Fraction Exponent?
Integer ::=
        MinimumInteger
    |   '#'
    |   '#' Integer
    |   '#' , Integer
MinimumInteger:
        0
        0 MinimumInteger
        0 , MinimumInteger
```

```
Fraction:
        MinimumFraction? OptionalFraction?
MinimumFraction:
        0 MinimumFractionopt
OptionalFraction:
        # OptionalFraction ?
Exponent:
        E MinimumExponent
MinimumExponent:
        0 MinimumExponent?
```

The pattern uses special characters:

- 0 matches all digits
- # matches all digits except the leading zeros
- . matches a decimal separator
- E separates mantissa and exponent in scientific notation.

Fixing the integer digits

```
<xsd:element name="year" type="xsd:int">
  <xsd:annotation>
    <xsd:appinfo>
      <sdf:block>
        <sdf:numericFormat pattern="0000"/>
        <sdf:length>6</sdf:length>
      </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

Example: 4 integer digits with leading zeros

0105 0075 1789

Fixing the fraction digits

```
<xsd:element name="observation" type="xsd:decimal">
  <xsd:annotation>
    <xsd:appinfo>
      <sdf:block>
        <sdf:numericFormat pattern="#.000"/>
        <sdf:length>8</sdf:length>
      </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

Example: 3 fraction digits

1010.000 200.057 50.128

Fixing the exponent digits

```
<xsd:element name="observation" type="xsd:decimal">
  <xsd:annotation>
    <xsd:appinfo>
      <sdf:block>
        <sdf:numericFormat pattern="0.000E00"/>
        <sdf:length>8</sdf:length>
      </sdf:block>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

Example: Scientific notation with 2 exponent digits

1.010E03 2.005E02 5.013E01

Fixing the positive prefix

```
xsd:element name="drift" type="xsd:double">
    <xsd:annotation>
      <xsd:appinfo>
        <sdf:block>
          <sdf:encoding>ASCII</sdf:encoding>
          <sdf:numericFormat pattern="+0.000000000000"/>
          <sdf:length>19</sdf:length>
        </sdf:block>
      </xsd:appinfo>
    </xsd:annotation>
</xsd:element>
```

Example: The sign is always present

+3.70139721781 -2.04636307899 +0.00000000000

Chapter 4: Using variable-content types

Signature definition

Syntax

```
<sdf:signature
  simpleContent: xs:string
</sdf:signature>
```

This node allows to select an element depending on its content. This node is necessary to handle elements with a variable content (choice group, type derivation, substitution group). In that case, the SDF engine will select the first substitution with a valid signature.

Handling unbounded occurrences

In case of unbounded occurrence, SDF will count every element that have a valid signature. Therefore, the last occurrence is never taken into account.

Using constrained occurrence

```
<xsd:complexType name="DATASET">
  <xsd:sequence>
    <xsd:element name="record" type="RECORD"
                 minOccurs="0" maxOccurs="unbounded">
      <xsd:annotation>
         <xsd:appinfo>
           <sdf:block>
             <sdf:signature>type!="EOF"</sdf:signature>
           </sdf:block>
          </xsd:appinfo>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="eof" type="EOF_RECORD"/>
  </xsd:sequence>
</xsd:complexType>
```

Handling choice group

Within a choice group, SDF checks the sub-particules in the document order and choose the first element that have a valid signature. This process is repeated for each occurrence.

Selecting an element in a choice group

```
<xsd:complexType name="HEADER_record"
              minOccurs="0" maxOccurs="unbounded">
  <xsd:choice>

   <xsd:element name="program" type="RINEX_program">
     <xsd:annotation>
       <xsd:appinfo>
         <sdf:block>
           <sdf:signature>label="PGM / RUN BY / DATE"</sdf:signature>
         </sdf:block>
       </xsd:appinfo>
     </xsd:annotation>
   </xsd:element>

   <xsd:element name="ion_alpha" type="RINEX_ion_alpha>
     <xsd:annotation>
        <xsd:appinfo>
          <sdf:block>
            <sdf:signature>label="ION ALPHA"</sdf:signature>
          </sdf:block>
        </xsd:appinfo>
      </xsd:annotation>
    </xsd:element>

  </xsd:choice>
</xsd:complexType>
```

Handling abstract types

Furthermore, SDF handles type derivations at run-time. It will instantiate the first derived type that matche its signature.

Selecting a derived type

```
<xsd:element name="GPS" type="RINEX"/>

<xsd:complexType name="RINEX_OBSERVATION">
  <xsd:annotation>
    <xsd:appinfo>
      <sdf:block>
        <sdf:signature>header/format/type="O"</sdf:signature>
      </sdf:block>
   </xsd:appinfo>
  </xsd:annotation>

  <xsd:complexContent>
    <xsd:extension base="RINEX">
        <!-- truncated -->
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="RINEX_NAVIGATION">
  <xsd:annotation>
    <xsd:appinfo>
      <sdf:block>
        <sdf:signature>header/format/type="N"</sdf:signature>
      </sdf:block>
   </xsd:appinfo>
  </xsd:annotation>

  <xsd:complexContent>
    <xsd:extension base="RINEX">
        <!-- truncated -->
    </xsd:extension>
```

```
    </xsd:complexContent>
  </xsd:complexType>
```

This is an application of the sdf:signature element to handle products with a variable format. In that case, SDF will recognize either an OBSERVATION product or a NAVIGATION product depending on the "header/format/type" value.

Table of Contents

Part I	**DRB XQuery engine v2.2**	**2**
Chapter 1	Introduction	3
Chapter 2	XQuery Basics	7
Chapter 3	Modules and prologs	11
Chapter 4	Primary Expressions	15
Chapter 5	Path Expressions	19
Chapter 6	Arithmetic Expressions	23
Chapter 7	Sequence Expressions	25
Chapter 8	Comparison Expressions	28
Chapter 9	Logical Expressions	29
Chapter 10	Element Constructors	30
Chapter 11	FLWOR Expressions	31
Chapter 12	Conditional Expressions	33
Chapter 13	Functions and Operators	34
Chapter 14	Mathematical functions	70
Chapter 15	DRB specific built-in functions	76
Chapter 16	XQuery conformance	81
Part II	**DRB XML-Schema 1.0 processor**	**84**
Chapter 1	Language overview	85
Chapter 2	XML-Schema basics	87
Chapter 3	Structure definition	91
Chapter 4	Datatype definition	95
Chapter 5	Built-in datatypes	98
Part III	**Structured Data Format**	**111**
Chapter 1	General overview	112
Chapter 2	Adding physical description	114
Chapter 3	Formatting atomic values	126
Chapter 4	Using variable-content types	132