XML Query Data Model

W3C Working Draft 11 May 2000

This version:
http://www.w3.org/TR/2000/WD-query-datamodel-20000511
Latest version:
http://www.w3.org/TR/query-datamodel
Editors:
Mary Fernandez (AT&T Labs) < mff@research.att.com >
Jonathan Robie (Software AG) <Jonathan.Robie@SoftwareAG-USA.com>

Copyright ©2000 W3C® (MIT, INRIA , Keio), All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply.

Abstract

This document defines the W3C XML Query Data Model, which is the foundation of the W3C XML Query Algebra; the XML Query Algebra will be specified in a future document. Together, these two documents [will] provide a precise semantics of the XML Query Language.

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. The latest status of this document series is maintained at the W3C.

This is a First Public Working Draft for review by W3C Members and other interested parties. It is a draft document and may be updated, replaced or made obsolete by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress". This is work in progress and does not imply endorsement by the W3C membership.

This document has been produced as part of the W3C XML Activity, following the procedures set out for the W3C Process. The document has been written by the XML Query Working Group.

The XML Query Working Group feels that the contents of this Working Draft are reasonably stable, and therefore encourages feedback.

Comments on this document should be sent to the W3C mailing list www-xml-query-comments@w3.org (archived at http://lists.w3.org/Archives/Public/www-xml-query-comments/).

A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR/.

Table of contents

1 Introduction
2 Concepts
- 2.1 Types and Signatures
  - 2.1.1 Reference Type
  - 2.1.2 Schema Types
- 2.2 Post Schema-Validated Infoset

3 Nodes
- 3.1 Data Model Instance
- 3.2 Document
- 3.3 Elements
- 3.4 Attributes
- 3.5 Namespaces
- 3.6 Processing Instructions
- 3.7 Comments
- 3.8 Values
- 3.9 Information Items

4 Example
5 Constraints on the Data Model
6 References

Appendices

A XML Information Set Mapping
- A.1 Notation and Pseudo-code Syntax
- A.2 Information Items
- A.3 Document Item
- A.4 Element Items
- A.5 Attribute Items
- A.6 Namespace Items
- A.7 Processing Instruction Items
- A.8 Comment Items
- A.9 Other Information Items

B Candidate Operators for XML Query Algebra
- B.1 Equality Operators
- B.2 Serialization Operators
- B.3 Order Operators
- B.4 Accessors for Reference Values

---

1 Introduction

This document defines the W3C XML Query Data Model, which is the foundation of the W3C XML Query Algebra; the XML Query Algebra will be specified in a future document. Together, these two documents [will] provide a precise semantics of the XML Query Language.

Several XML data models have been developed in the W3C Activities. The XML Information Set (http://www.w3.org/TR/xml-infoset) provides a description of the information available in a well-formed XML document. The XPath Recommendation (http://www.w3.org/TR/xpath), which is used by both XSLT (http://www.w3.org/TR/xslt) and XPointer (http://www.w3.org/TR/xptr), contains a data model and a mapping that relates the XPath data model to the XML Information Set (hence ‘‘Infoset’’). The Document Object Model (http://www.w3.org/TR/DOM-Level-2/) is an API for HTML and XML documents, but it does imply an underlying abstract data model. The XML Schema Working Group is defining features, such as structures (http://www.w3.org/TR/xmlschema-1) and datatypes (http://www.w3.org/TR/xmlschema-2), that extend an instance of the XML Information Set with more precise type information.

The XML Query Data Model defines formally the information contained in the input to an XML Query processor; in other words, an XML Query processor evaluates a query on an instance of the XML Query Data Model. Our model is based on the XML Information Set, but it requires the following new features to meet the XML Query Working Group’s requirements:

- Support for XML Schema types. (http://www.w3.org/TR/xmlschema-1, http://www.w3.org/TR/xmlschema-2)
- Representation of Collections of Documents and of Simple and Complex Values. (http://www.w3.org/TR/2000/WD-xmlquery-req-20000131#collections)
- Representation of References. (http://www.w3.org/TR/2000/WD-xmlquery-req-20000131#references)

In this document, we provide a precise and formal definition of how values in the XML Query Data Model are constructed and accessed; these operators are the foundation of the XML Query Algebra, and therefore, require a more formal treatment than is provided in the definitions of the XPath and Infoset data models. For comparison, we note wherever the XML Query Data Model differs from that of XPath, and we provide a formal definition of the mapping from the Infoset to the XML Query Data Model in Appendix A.

An instance of the XML Query data model represents one or more complete XML documents or document parts. As with the Infoset, the XML Query Data Model specifies what information in the documents is accessible, but it does not specify the programming-language interfaces or bindings used to represent or access the data. We assume that the information contained in an instance of the XML Query Data Model is static for the duration query evaluation; in particular, the meaning of a query applied to input data that changes during query evaluation is undefined.

We define the data model using a functional notation. We chose this notation because it is simple and permits a precise and concise definition of the data model. The notation is presented in the next section. Although the notation has a functional style, we emphasize that the data model can be realized in a variety of programming languages and styles, for example, as object classes and methods in an object-oriented language.

There is a close correspondence between the data model and the algebraic operations that are applied to instances of the data model. In the definition of the data model, we mention several algebraic operators. These and other candidate operators are identified in the Candidate Operators for XML Query Algebra appendix. The complete algebra will be defined in a future document.

2 Concepts

Because XML documents are tree-structured, we define the XML Query data model using conventional terminology for trees. Various mathematical representations of trees exist (see [CLR]). The *graph-theoretic* representation models a tree as a tuple *(N, E, r)*, where *N* is a set of tree nodes, *E* is a one-to-many relation of edges from parent nodes to children nodes, and *r* is the distinguished root node of the tree. The *tree-constructor* representation models a tree by recursive application of a tree-creation function to a list of children trees [Knuth]. In either representation, a tree can have data associated with its nodes (*node-labeled*), with its edges (*edge-labeled*), or with both. The XPath and Infoset data models are examples of node-labeled, tree-constructor representations.

The XML Query Data Model (hence ‘‘data model’’) is a node-labeled, tree-constructor representation, but also includes a concept of node identity. The addition of node identity simplifies the representation of XML reference values, e.g., IDREF, XPointer, and URI values. Unlike the graph model, the data model does not support both nodes and edges as first-class concepts. We note, however, whenever a graph model may be more flexible or concise.

2.1 Types and Signatures

The data model defines the structure of various kinds of tree nodes; functions to construct tree nodes, called *constructors*; and functions to access nodes' structure, called *accessors*. This presentation is similar to that in the definition of the XSLT data model [Wadler].

We use the terms *node type* to refer to the kind of a tree node; *schema type* to refer to an XML Schema type; and *collection type* to refer to tuples, lists, sets, and disjoint unions. The term *type* refers to any node, schema, or collection type. The term *signature* refers to a constructor's or accessor's function type, i.e., the function's zero or more input types and its one output type.

To make the definition of the data model concise, we use the following notation:

- *N* ranges over *node types*.
- *U* ranges over the union of node types, schema types, and collection types.
- *[U]* denotes a list of values with type *U*; a list is ordered and may contain duplicates.
- *{ U }* denotes a set of values with type *U*; a set is unordered and does not contain duplicate values.
- *{| U |}* denotes a bag of values with type *U*; a bag is unordered and may contain duplicate values.

- *(U1, U2, ..., Un)* denotes a tuple whose first component has type *U1*, second component has type *U2*, etc.

- *U1 | U2 | ... | Un* denotes the disjoint union of values with types *U1*,..., *Un*, etc.

- *TN = U* denotes that the string *TN* is a type name that represents the type *U*. A type name may appear wherever its associated type is expected.

An occurrence of the type symbol *U* denotes a *value* or *instance* of that type.

The signatures of constructors and accessors use the following notation:

- *f : U1 -> U2* denotes a function *f* that takes an input value with type *U1* and returns an output value with type *U2*.

2.1.1 Reference Type

The data model provides node references as a mechanism to test and bind the identity of nodes in a given instance of the data model. The actual mechanism for implementing node identity is implementation dependent, for example, node identity might be represented by a key value, an object identifier, an XPointer value, etc.

- *Ref(N)* denotes a reference to a node with type *N*.

The data model provides the function *ref* to create a reference to a node and the function *deref* to produce the node referent of a reference value. Their signatures are:

```
ref          : Node      -> Ref(Node)
deref        : Ref(Node) -> Node
```

The function *ref* is surjective, i.e., it is onto. The function *deref* is the inverse of *ref*, i.e., for all nodes *n*, *node_equal(deref(ref(n)), n)* is true, where *node_equal* is an implementation-dependent equality operator over nodes.

Some accessors map values (e.g., IDREF or key values) into reference values. If a key value does not represent a valid reference, the "not a reference" value, *NaR*, may be used.

2.1.2 Schema Types

Nodes contain values that are instances of schema types. The following symbols denote schema types:

- *T* ranges over XML schema types, which are either simple or complex.

- *ST* ranges over simple types.

- *CT* ranges over complex types.

A simple type is either *primitive* (e.g., *string, boolean, float, double, ID, IDREF*) or *derived* (e.g., *language, NMTOKEN, long*, etc., or user defined). A complex type defines the structure and content of an element.

It may be necessary for a query to have access to a value's type during query execution, i.e., at run time. Because an instance of an XML Schema is also an XML document (see XML Representation of Schemas and Schema Components in [XMLSchema Part 1]), a document's schema can be represented as an instance of the data model.

- *Def_T = ElemNode* denotes the data model representation of schema type *T*.

Because *Def_T* is an element value in the data model, i.e., an *ElemNode*, all of the accessors defined for *ElemNode* can be applied to *Def_T* values.

2.2 Post Schema-Validated Infoset

We assume that an instance of the data model is derived from an instance of the XML Information Set [XML Infoset] after XML Schema validation. Such an instance is called a ''PSV Infoset'', for post schema-validated Infoset.

A well-formed document may have an associated schema, derived from one or more XML Schema documents; it may have an associated DTD; or it may have no schema, i.e., it is ''schemaless''. For a document with a schema, the PSV Infoset provides the schema. For a document with a DTD, we assume a mapping from the DTD to an equivalent schema is provided. For a schemaless document, a default schema is provided by the data model, and we specify in this document that default schema. In all three cases, a schema can be provided for the input document and corresponding data-model instance.

3 Nodes

The basic concept in the data model is a *Node*. A *Node* is one of eight *node types*: document, element, value, attribute, namespace (NS), processing instruction (PI), comment, or information item. This definition specifies that a *Node* is the disjoint union of eight node types:

```
Node =   DocNode | ElemNode | ValueNode   | AttrNode
       | NSNode  | PINode   | CommentNode | InfoItemNode
```

The eight node types are defined in the following subsections. Note that an implementation of the data model in an object-oriented language might choose to make *Node* an interface (or an abstract class) and to make each node type a concrete class that implements the *Node* interface (or a sub-class of the abstract class *Node*).

A *Node* has eight accessors, each of which returns true if the *Node* value is of the specified type.

```
isDocNode       : Node -> boolean
isElemNode      : Node -> boolean
isValueNode     : Node -> boolean
isAttrNode      : Node -> boolean
isNSNode        : Node -> boolean
isPINode        : Node -> boolean
```

```
isCommentNode  : Node -> boolean
isInfoItemNode : Node -> boolean
```

3.1 Data Model Instance

An XML document is represented by its distinguished document node, which is a *DocNode*. A *document part* is a sub-tree of a document and is represented by an *ElemNode*, *ValueNode*, *PINode*, or *CommentNode*. An instance of the data model represents one or more complete documents or document parts and may be unordered or ordered. Therefore, a data-model *Instance* is a set, a list, or a bag of one or more nodes:

```
Instance = { DocNode | ElemNode | ValueNode | PINode | CommentNode }
         | [ DocNode | ElemNode | ValueNode | PINode | CommentNode ]
         | {| DocNode | ElemNode | ValueNode | PINode | CommentNode |}
```

3.2 Document

A document is represented by a unique *DocNode*, which corresponds to a document information item in the Infoset. A *DocNode* has the constructor *docNode*, which takes a URI reference value and a non-empty list of its children nodes:

```
docNode   : (URIRefValue, [ Ref(ElemNode) | Ref(PINode) | Ref(CommentNode) ])
          -> DocNode
```

The list of children nodes may contain zero or more references to *PINode*s and *CommentNode*s and must contain exactly one reference to an *ElemNode*.

The accessor *uri* returns a *DocNode*'s URI reference value, and the accessor *children* returns a *DocNode*'s list of children:

```
uri      : DocNode -> URIRefValue
children : DocNode -> [ Ref(ElemNode) | Ref(PINode) | Ref(CommentNode) ]
```

The *root* accessor returns a reference to the unique *ElemNode* in a *DocNode*'s children list:

```
root     : DocNode -> Ref(ElemNode)
```

3.3 Elements

An *ElemNode* has the constructor *elemNode*, which takes a tag value, a set of *NSNode* namespaces, a set of *AttrNode* attributes, a list of children nodes, and a reference to the node's type:

```
elemNode : (Ref(QNameValue), { Ref(NSNode) }, { Ref(AttrNode) },
            [ Ref(ElemNode) | Ref(ValueNode) | Ref(PINode)
            | Ref(CommentNode) | Ref(InfoItemNode) ],
            Ref(Def_T))
         -> ElemNode
```

An *ElemNode*'s tag is a qualified name value, *QNameValue*. An *ElemNode*'s set of namespaces contain one namespace node for each distinct namespace that is declared explicitly on the element. The node's children is a list of references to *ElemNode*, *ValueNode*, *PINode*,

CommentNode, and *InfoItemNode* values.

We assume that the element is an instance of the type represented by *Def_T*, i.e., the document "type checks" or is valid with respect to the given schema. For a schemaless document, the data model defines a default schema; in this case, the *Def_T* value for all *ElemNode*s is the ur-type definition [XMLSchema Part 1].

The accessors *name*, *namespaces*, *attributes*, *children*, and *type* returns an *ElemNode*'s constituent parts:

```
name       : ElemNode -> Ref(QNameValue)
namespaces : ElemNode -> { Ref(NSNode) }
attributes : ElemNode -> { Ref(AttrNode) }
children   : ElemNode -> [ Ref(ElemNode) | Ref(ValueNode) | Ref(PINode)
                         | Ref(CommentNode) | Ref(InfoItemNode) ]
type       : ElemNode -> Ref(Def_T)
```

The accessor *parent* returns a reference to the unique parent of an *ElemNode*. If an *ElemNode* is the root node of a document, *parent* will return a reference to the corresponding *DocNode*, unless a document node does not exist, in which case *parent* returns *NaR*, the "not a reference" value.

```
parent     : ElemNode -> Ref(ElemNode) | Ref(DocNode) | NaR
```

> **NOTE:** The XPath data model does not model the complex type of a node. This definition adds *Def_T* to an element node, and it also adds *ValueNode* and *InfoItemNode* as permissible children of an element node.

3.4 Attributes

An *AttrNode* has the constructor *attrNode*, which takes the attribute's name and value:

```
attrNode : (Ref(QNameValue), Ref(ValueNode)) -> AttrNode
```

The accessors *name* and *value*, return an attribute's constituent parts:

```
name     : AttrNode  -> Ref(QNameValue)
value    : AttrNode  -> Ref(ValueNode)
```

For a schemaless document, the data model defines a default schema; in this case, the type of all *AttrNode*'s values is the simple type

```
<simpleType base="string"/>.
```

The accessor *parent* returns a reference to the containing element of an *AttrNode*.

```
parent   : AttrNode -> Ref(ElemNode)
```

> **NOTE:** The XPath data model only permits attribute values to be strings. This definition permits attribute values to be of any simple type.

3.5 Namespaces

An *NSNode* has the constructor *nsNode*, which takes an namespace prefix, which may be null, and the absolute URI of the namespace being declared, which may be null:

```
nsNode : (Ref(StringValue) | null, Ref(URIRefValue) | null) -> NSNode
```

The accessors *prefix* and *uri* return a *NSNode*'s constituent parts:

```
prefix : NSNode -> Ref(StringValue) | null
uri    : NSNode -> Ref(URIRefValue) | null
```

A namespace node may contain: a non-null prefix and a non-null uri; a null prefix and a non-null uri; or a null prefix and a null uri. A namespace node may not contain a non-null prefix and a null uri.

The accessor *parent* returns a reference to the containing element of an *NSNode*.

```
parent : NSNode -> Ref(ElemNode)
```

3.6 Processing Instructions

A *PINode* has the constructor *piNode*, which takes a local name that is the processing instruction's target and a string value:

```
piNode : (Ref(StringValue), Ref(StringValue)) -> PINode
```

The local name is a string value that must have type NCNAME, i.e., its *type* value refers to the definition of the derived type NCName :

```
<simpleType name="NCName" base="Name"/>
```

The accessors *target*, *value* return a *PINode*'s constituent parts.

```
target : PINode -> Ref(StringValue)
value  : PINode -> Ref(StringValue)
```

The accessor *parent* returns a *PINode*'s unique parent:

```
parent : PINode -> Ref(ElemNode) | Ref(DocNode)  | NaR
```

3.7 Comments

A *CommentNode* has the constructor *commentNode*, which takes a string value:

```
commentNode : Ref(StringValue) -> CommentNode
```

The accessor *value* returns a *CommentNode*'s value, and the accessor *parent* returns a *CommentNode*'s unique parent:

```
value     : CommentNode -> Ref(StringValue)
parent    : CommentNode -> Ref(ElemNode) | Ref(DocNode) | NaR
```

3.8 Values

A *ValueNode* is the disjoint union of fourteen kinds of simple-type values; each value kind has an associated constructor.

```
ValueNode = StringValue  | BoolValue     | FloatValue    | DoubleValue
          | DecimalValue | TimeDurValue  | RecurDurValue | BinaryValue
          | URIRefValue  | IDValue       | IDREFValue    | QNameValue
          | ENTITYValue  | NOTATIONValue
```

The data model provides constructors for the fourteen primitive XML Schema datatypes:

```
stringValue  : (string, Ref(Def_string), [Ref(InfoItemNode)]) -> StringValue
boolValue    : (boolean, Ref(Def_boolean))                    -> BoolValue
floatValue   : (float, Ref(Def_float))                        -> FloatValue
doubleValue  : (double, Ref(Def_double))                      -> DoubleValue
decimalValue : (decimal, Ref(Def_decimal))                    -> DecimalValue
timeDurValue : (timeDuration, Ref(Def_TimeDuration))          -> TimeDurValue
recurDurValue: (recurringDuration, Ref(Def_recurringDuration))-> RecurDurValue
binaryValue  : (binary, Ref(Def_binary))                      -> BinaryValue
urirefValue  : (uriReference, Ref(Def_uriReference))          -> URIRefValue
idValue      : (ID, Ref(Def_ID))                              -> IDValue
idrefValue   : (IDREF, Ref(Def_IDREF))                        -> IDREFValue
qnameValue   : (uriReference | null, string, Ref(Def_QName))  -> QNameValue
entityValue  : (ENTITY, Ref(Def_ENTITY))                      -> ENTITYValue
notationValue: (NOTATION, Ref(Def_NOTATION))                  -> NOTATIONValue
```

Ed. Note:

We note here that the data model does not currently represent key values and key reference values as described in XML Schema Part 1 : Structures [XMLSchema Part 1]. In a future draft of this document, keys and key references will be represented in the data model.

Derived datatypes are modeled as instances of their primitive base type. In particular, wherever an instance of *Def_ST* is expected, we assume an instance of *ST'* is permitted if and only if *ST'* is derived from simple type *ST*. For example, the derived simple type *long* has base type *integer*, so an instance of *long* can be modeled by an *IntegerValue* whose type is *Ref(Def_long)*, i.e., it refers to the definition of the derived type long:

```
<simpleType name="long" base="integer"/>
```

The explicit enumeration of each kind of value guarantees that the type of the value's instance and its type representation correspond. For example, a *StringValue* always contains a *string* value and a reference to the representation of a primitive or derived type whose base type is *string*. For example, a product's ‘‘Sku’’ number could be represented by the string value:

```
StringValue("926-AA", Ref(Def_Sku), [])
```

where the Sku type is derived from string:

```
<simpleType name="Sku" base="string">
  <pattern value="\d{3}-[A-Z]{2}"/>
</simpleType>
```

The accessor *type* returns a *ValueNode*'s type:

```
type       : ValueNode    -> Ref(Def_ST)
```

The accessor *string* returns a *ValueNode*'s string representation:

```
string     : ValueNode    -> StringValue
```

The accessor *infoItems* returns the list of information items associated with a *StringValue*; this list may contain the character information items from which the string value was created:

```
infoItems : StringValue -> [ Ref(InfoItemNode) ]
```

The accessors *localPart* and *uriPart* return a *QNameValue*'s constituent parts:

```
uriPart   : QNameValue  -> uriReference | null
localPart : QNameValue  -> string
```

The accessor *referent* returns a list of element-node references associated with an *IDREFValue* or *URIRefValue*:

```
referent  : (IDREFValue | URIRefValue) -> [ Ref(ElemNode) | NaR ]
```

If a referent is not in the data-model instance, *referent* returns *NaR*, the "not a reference" value. In a data-model instance of a PSV infoset, every IDREF and keyref value is guaranteed to refer to a *ElemNode* in the data-model instance. This may not be the case for a URI value, which could refer to an element in an arbitrary document that is not contained in the data-model instance. In this case, *referent* may return a list containing *NaR*.

A *ValueNode* has fifteen additional accessors, each of which returns true if the *ValueNode* is of the specified type.

```
isStringValue   : ValueNode -> boolean
isBoolValue     : ValueNode -> boolean
isFloatValue    : ValueNode -> boolean
isDoubleValue   : ValueNode -> boolean
isDecimalValue  : ValueNode -> boolean
isTimeDurValue  : ValueNode -> boolean
isRecurDurValue : ValueNode -> boolean
isBinaryValue   : ValueNode -> boolean
isURIRefValue   : ValueNode -> boolean
isIDValue       : ValueNode -> boolean
isIDREFValue    : ValueNode -> boolean
isQNameValue    : ValueNode -> boolean
isENTITYValue   : ValueNode -> boolean
isNOTATIONValue : ValueNode -> boolean
```

NOTE: *ValueNode* replaces the XPath data model's text node.

3.9 Information Items

An *InfoItemNode* has the constructor *infoItemNode*:

```
infoItemNode : Ref(InfoItem) -> InfoItemNode
```

An *InfoItemNode* is an opaque value that preserves those information items that are not of interest to the data model or query language, but are required by the underlying Infoset implementation. It has no accessors.

4 Example

We use the following XML document to illustrate the information contained in an instance of the data model:

```
<?xml version=1.0?>
<p:part xmlns:p="http://www.mywebsite.com/PartSchema"
      xsi:schemaLocation = "http://www.mywebsite.com/PartSchema
                            http://www.mywebsite.com/PartSchema"
      name="nutbolt">
  <mfg>Acme</mfg>
  <price>10.50</price>
</p:part>
```

The XML schema for this document is:

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  targetNamespace="http://www.mywebsite.com/PartSchema">
  <xsd:element name="part" type="part_type">
    <xsd:complexType name="part_type">
      <xsd:element name = "mfg" type="xsd:string"/>
      <xsd:element name = "price" type="xsd:decimal"/>
      <xsd:attribute name = "name" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

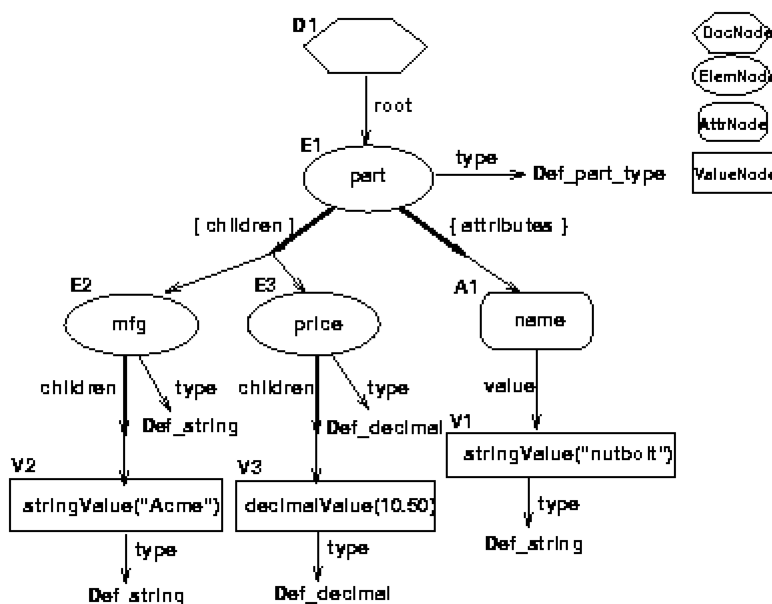
For this example, we chose a XML document that includes an XML Schema to illustrate the relationship between document content and its associated schema type information. In general, an XML Schema is not required, that is, the data model can represent a schemaless, well-formed XML document with the default typing described in this document.

The XML document is represented by the data-model accessors below. The value *D1* represents a *DocNode*; the values *E1, E2, etc.* represent *ElemNode*s; the values *A1, etc.* represent *AttrNode*s; the values *N1, etc.* represent *NSNode*s.

```
children(D1)   = [ Ref(E1) ]
root(D1)       = Ref(E1)

name(E1)       = QNameValue("http://www.mywebsite.com/PartSchema",
                            "part", Ref(Def_QName))
children(E1)   = [ Ref(E2), Ref(E3) ]
attributes(E1) = { Ref(A1) }
namespaces(E1) = { Ref(N1) }
```

```
type(E1)        = Ref(Def_part_type)
parent(E1)      = Ref(D1)

name(A1)        = QNameValue(null, "name", Ref(Def_QName))
value(A1)       = Ref(StringValue("nutbolt", Ref(Def_string)))
parent(A1)      = Ref(E1)

prefix(N1)      = Ref(StringValue("p", Ref(Def_string)))
uri(N1)         = URIRefValue("http://www.mywebsite.com/PartSchema",
                              Ref(Def_uriReference))
parent(N1)      = Ref(E1)

name(E2)        = QNameValue(null, "mfg", Ref(Def_QName))
children(E2)    = [ Ref(StringValue("Acme", Ref(Def_string))) ]
attributes(E2) = {}
namespaces(E2) = {}
type(E2)        = Ref(Def_string)
parent(E2)      = Ref(E1)

name(E3)        = QNameValue(null, "price", Ref(Def_QName))
children(E3)    = [ Ref(DecimalValue(10.50, Ref(Def_decimal))) ]
attributes(E3) = {}
namespaces(E3) = {}
type(E3)        = Ref(Def_decimal)
parent(E3)      = Ref(E1)
```

A graphical depiction of the data-model instance, containing the information described in the text above, is also included.



Recall that an XML schema is also an XML document, which can be represented in the data model. The complex and simple types in the example schema are represented below:

```
name(Def_part_type)        = QNameValue("http://www.w3.org/1999/XMLSchema",
                                        "complexType", Ref(Def_QName))
children(Def_part_type)    = [ Ref(E4), Ref(E5), Ref(E6) ]
attributes(Def_part_type) = { Ref(A2) }
namespaces(Def_part_type) = {}
```

```
type(Def_part_type)       = Ref(Def_complexType)

name(A2)        = QNameValue(null, "name", Ref(Def_QName))
value(A2)       = Ref(StringValue("part_type", Ref(Def_NCName), []))

name(E4)        = QNameValue("http://www.w3.org/1999/XMLSchema",
                             "element", Ref(Def_QName))
children(E4)    = []
attributes(E4) = { Ref(A3), Ref(A6) }
namespaces(E4) = {}
type(E4)        = Ref(Def_element)

name(A3)        = QNameValue(null, "name", Ref(Def_QName))
value(A3)       = Ref(StringValue("mfg", Ref(Def_NCName), []))

name(A6)        = QNameValue(null, "type", Ref(Def_QName))
value(A6)       = Ref(QNameValue("http://www.w3.org/1999/XMLSchema",
                                 "string", Ref(Def_QName), []))

name(E5)        = QNameValue("http://www.w3.org/1999/XMLSchema",
                             "element", Ref(Def_QName))
children(E5)    = []
attributes(E5) = { Ref(A4), Ref(A7) }
namespaces(E5) = {}
type(E5)        = Ref(Def_element)

name(A4)        = QNameValue(null, "name", Ref(Def_QName))
value(A4)       = Ref(StringValue("price", Ref(Def_NCName)))

name(A7)        = QNameValue(null, "type", Ref(Def_QName))
value(A7)       = Ref(QNameValue("http://www.w3.org/1999/XMLSchema",
                                 "decimal", Ref(Def_QName), []))

name(E6)        = QNameValue("http://www.w3.org/1999/XMLSchema",
                             "attribute", Ref(Def_QName))
children(E6)    = []
attributes(E6) = { Ref(A5), Ref(A8) }
namespaces(E6) = {}
type(E6)        = Ref(Def_attribute)

name(A5)        = QNameValue(null, "name", Ref(Def_QName))
value(A5)       = Ref(StringValue("name", Ref(Def_NCName), []))

name(A8)        = QNameValue(null, "type", Ref(Def_QName))
value(A8)       = Ref(QNameValue("http://www.w3.org/1999/XMLSchema",
                                 "string", Ref(Def_QName), []))
```

For conciseness, we eliminate the definitions of builtin schema components, e.g., *complexType*, and of primitive simple types, e.g., *string*.

5 Constraints on the Data Model

A data-model instance must satisfy the following constraints:

- Node references: We assume that the representation of a node reference is defined by the query system that implements the data model, not by the query language itself. Node references are not serialized, i.e., they exist only for use by the query system, and they are not guaranteed to be globally unique or persistent, although some implementation of the data model may choose to support persistent node references. Multiple techniques may exist for implementing references, and there may be multiple techniques for implementing

identity.

- Node identity: the function *ref* is one-to-one and onto, i.e., *ref_equal(ref(n1), ref(n2))* holds if and only if *n1* and *n2* are the same node. We assume that the operator *ref_equal* is an implementation-dependent equality operator over node references.

- Unique parent: the *parent* accessor is a many-to-one function, i.e., a node has exactly one parent.

- Parent-child relationships: Given two *ElemNode* references *ref(p)* and *ref(n)*, *ref_equal (parent(n), ref(p))* holds if and only if *ref(n)* is in *children(p)*.

  Similarly, given a *DocNode* reference *ref(d)* and a node reference *ref(n)*, *ref_equal(parent (n), ref(d))* if and only if *ref(n)* is in *children(d)*.

  Finally, given a *AttrNode* (or *NSNode*) reference *ref(a)* and a node reference *ref(n)*, *ref_equal(parent(a), ref(n))* if and only if *ref(a)* is in *attributes(n)* (*namespaces(n)*).

- Duplicate-free list of children: Given a node *n* and any two node references *r1* and *r2* at distinct positions in *children(n)*, *not ref_equal(r1, r2)* must hold i.e., the ordered list of children nodes is duplicate free.

6 References

CLR
T. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, MIT Press, 1991.
Knuth
D. Knuth, *The Art of Computer Programming, Vol. 1*, Addison-Wesley, 1973.
Wadler
P. Wadler, *A formal semantics of patterns in XSLT*. Available at: http://www.cs.bell-labs.com/who/wadler/papers/xsl-semantics/xsl-semantics.pdf
XML Infoset
World Wide Web Consortium, *XML Information Set (Infoset)*. Available at: http://www.w3.org/TR/xml-infoset
XPath
World Wide Web Consortium, *XML Path Language (XPath)*. Available at: http://www.w3.org/TR/xpath.html
XMLSchema Part 1
World Wide Web Consortium, *XML Schema Part 1 : Structures*. Available at: http://www.w3.org/TR/xmlschema-1
XMLSchema Part 2
World Wide Web Consortium, *XML Schema Part 2 : Datatypes*. Available at: http://www.w3.org/TR/xmlschema-2

A XML Information Set Mapping

The XML Information Set (http://www.w3.org/TR/xml-infoset) provides a description of the

information available in a well-formed XML document. In this appendix, we define the Infoset in the same notation as the data model, and we define functions that map from an instance of the Infoset to an instance of the data model in a pseudo-code notation.

> **Ed. Note:** In a future draft of this document, a mapping from the XML Query data model into the Infoset will be provided.

A.1 Notation and Pseudo-code Syntax

We name accessors of the PSV Infoset using the convention *psv_<item-name>_<property>* . For example, *psv_elem_attributes* is the accessor that returns an element item's attributes property. The comments in the accessor definitions specify whether a property is *core* or *peripheral*.

The following operations are defined on lists and are used in the function definitions:

- *[]* constructs the empty list.
- *[n]* constructs a list with one element *n*.
- *h :: t* constructs a list with head element *h* and tail list *t*.

It is often necessary to deconstruct a list value, for example, into its head element and tail list. We use the same notation to match and deconstruct list values. For example, this function maps all *"a"* values in a list to *"b"* values by deconstructing and matching the argument list *L*:

```
fun a2b(L) {
 case L of []           => []
         |  [ "a" ] :: t => [ "b" ] :: (a2b t)
         |  h :: t       => h :: (a2b t)
}
a2b [ "a", "c", "a" ] = [ "b", "c", "b" ]
```

The *case* expression matches its list argument *L* against each list pattern and evaluates the right-hand-side of the first pattern that matches *L*.

The function *concat_list* concatenates two lists, and *concat_string* concatenates two strings:

```
concat_list : ([ U ], [ U ]) -> [ U ]
concat_string : (string, string) -> string
```

For example,

```
concat_list [ 1, 2 ] [ 3, 4 ] = [ 1, 2, 3, 4 ]
concat_string "First" "Last" = "FirstLast"
```

The function *flat_list* takes a list of lists and returns the flattened list:

```
flat_list : [ [ U ] ] -> [ U ]
```

For example,

```
flat_list([ [1], [2, 3] ]) = [ 1, 2, 3 ]
```

The function *map_set* applies a function to every member of a set and returns the transformed set; *map_list* is analogous for lists:

```
map_set  : ((U1 -> U2), { U1 }) -> { U2 }
map_list : ((U1 -> U2), [ U1 ]) -> [ U2 ]
```

For example, given the function *addOne*:

```
fun add_one(x) { return x + 1 }
map_set  add_one { 1, 2 } = { 2, 3 }
map_list add_one [ 1, 2 ] = [ 2, 3 ]
```

A.2 Information Items

The basic concept in the Infoset is an *InfoItem*. An *InfoItem* is one of ten item types:

```
InfoItem = DocItem        | ElemItem      | CDATAItem    | AttrItem
         | NSItem         | PIItem        | CommentItem  | SkipEntityItem
         | EntityMarker   | CDATAMarker
```

No constructors are defined for *Item* values, because they are constructed by an Infoset processor, not the data model.

A.3 Document Item

The following accessors are defined on a document information item:

```
/* core */
psv_doc_children : DocItem -> [ /* core */
                                Ref(PIItem) | Ref(ElemItem)
                                /* peripheral */
                              | Ref(CommentItem) | Ref(DocTypeItem) ]
psv_doc_notations : DocItem-> { Ref(NotationItem) }
psv_doc_base_uri  : DocItem -> uriReference

/* core : unparsed entities; peripheral : parsed entities */
psv_doc_entities  : DocItem -> { Ref(EntityItem) }
```

Exactly one *ElemItem* is required in the *psv_doc_children* list and it is the root element of the document tree.

The following pseudo-code documents the mapping from an instance of the PSV infoset to an instance of the data model. The function *dm_docNode* maps a document information item (*DocItem*) to a document node (*DocNode*):

```
dm_docNode : DocItem -> DocNode
fun dm_docNode(d) {
  kids = map_list dm_node (psv_doc_children d)
  return docNode(uriValue(psv_doc_base_uri d, ref(Def_uriReference)), kids)
}
```

In the definition of *kids*, *map_list* applies the function *dm_node* to each child of the *DocItem* value *d*; this constructs a new list of children nodes, each of which has type *Node*. The application of the constructor *docNode* constructs the document node in the data model.

A.4 Element Items

The following accessors are defined on element information items:

```
/* core */
psv_elem_uri       : ElemItem -> uriReference
psv_elem_local_name: ElemItem -> string

psv_elem_children  : ElemItem -> [ /* core */
                                   Ref(PIItem)
                                 | Ref(ElemItem)
                                 | Ref(SkipEntityItem)
                                 | Ref(CDATAItem)

                                   /* peripheral */
                                 | Ref(CommentItem)
                                   /* start, end marker pairs : */
                                 | (Ref(EntityMarker), Ref(EntityMarker))
                                 | (Ref(CDATAMarker), Ref(CDATAMarker))
                                 ]
/* core */
psv_elem_attrs       : ElemItem -> { AttrItem }
psv_elem_decl_ns     : ElemItem -> { NSItem } /* declared namespaces */
psv_elem_base_uri    : ElemItem -> uriReference
psv_elem_parent      : ElemItem -> (Ref(ElemItem) | Ref(DocItem))

/* Infoset item for schema type */
psv_elem_schema_type : ElemItem -> Ref(ElemItem)

/* peripheral */
psv_elem_inscope_ns  : ElemItem -> { NSItem } /* in-scope namespaces */
```

The function *dm_elemNode* maps an *ElemItem* to an *ElemNode*:

```
dm_elemNode : ElemItem -> ElemNode
fun dm_elemNode(e) {
  name      = qnameValue(psv_elem_uri e, psv_elem_local_name e, Ref(Def_QName))
  nsnodes   = map_set dm_nsNode (psv_elem_decl_ns e)
  attrnodes = map_set dm_attrNode (psv_elem_attrs e)
  kids      = dm_collapse_strings(map_list dm_node (psv_elem_children e))
  type      = dm_schema_type(psv_elem_schema_type e)
  newkids   = if (isSimpleType(type)) (map_list (dm_coerce_string type) kids)
              else kids
  return elemNode(name, nsnodes, attrnodes, newkids, type)
}
```

This function extracts components from the *ElemItem*, transforms them into instances of data-model values, and then constructs an *ElemNode* value.

The function *dm_schema_type* returns a reference to the *Def_T* value that corresponds to the schema type represented by its information-item argument.

```
dm_schema_type    : Ref(ElemItem) -> Ref(Def_T)
```

We assume above that if an element's type is a simple type, its list of children must contain one string value. The function *dm_coerce_string* coerces its string-valued second argument into an instance of the simple type referenced by its first argument.

```
dm_coerce_string : (Ref(Def_ST), StringValue) -> ST
```

Ed. Note: This mapping does not capture an element item's keyref values, which are maintained in the element item's identity constraints table. The identity constraints table is created for ‘‘bookkeeping purposes’’ by an XML Schema processor (see [XMLSchema Part 1]). The mapping of an element item's identity constraints table into keyref values will be specified in a future draft of this document.

The function *dm_node* maps an information item to a reference to a data-model node. All information items are preserved in the data model, but the document type, skipped entity reference, entity markers, and character data markers are represented by the opaque data type *InfoItemNode*.

```
dm_node : InfoItem -> Ref(Node)
fun dm_node(i) {
  case i of
    Ref(PIItem e)             => ref(dm_piNode e)
  | Ref(ElemItem e)           => ref(dm_elemNode e)
  | Ref(CommentItem)          => ref(dm_commNode e)
  | Ref(CharDataItem e)       => ref(dm_char2string e)
  | r as Ref(DocTypeItem)     => ref(infoItemNode r)
  | r as Ref(SkipEntityItem) => ref(infoItemNode r)
  | p as (Ref(EntityMarker), Ref(EntityMarker)) => ref(infoItemNode p)
  | p as (Ref(CDATAMarker),  Ref(CDATAMarker))  => ref(infoItemNode p)
}
```

The function *dm_char2string* takes one CDATA item and maps it to a *StringValue* with a string value of length one. Note that the *StringValue* constructor preserves the original CDATA item in a *InfoItemNode*.

```
dm_char2string : CDATAItem -> StringValue
fun dm_char2string(c) {
  /* convert (psv_cdata_code c) to string of length 1 */
  stringValue(code2string(psv_cdata_code c), Ref(Def_string),
              [infoItemNode (ref c)])
}
```

The function *dm_collapse_strings* synthesizes a single string value from multiple CDATA items. It does this by collapsing one or more consecutive *StringValues* in its argument list, each of which must have the simple schema type *string*, into one *StringValue* whose content is the concatenation of the values of the consecutive *StringValues*. All other node values are returned unchanged.

```
dm_collapse_strings : [ Ref(Node) ] -> [ Ref(Node) ]
fun dm_collapse_strings(nodelist) {
  case nodelist of :
    []      => return []
  | [ n ]   => return [ n ]
  | h1 as Ref(StringValue(s1, Ref(Def_string), i1)) :: t1 =>
    { case t1 of :
        []    =>   return [ h1 ]
```

```
        /* Collapse two consecutive strings and
           apply dm_collapse_strings recursively */
      | Ref(StringValue(s2, Ref(Def_string), i2)) :: t2 =>
          return dm_collapse_strings(
                   ref(stringValue(concat_string(s1, s2),
                                   Ref(Def_string),
                                   concat_list(i1, i2))) :: t2)
      | h :: t => return (h :: (dm_collapse_strings t))
    }
}
```

A.5 Attribute Items

The following accessors are defined for attribute information items:

```
/* core */
psv_attr_children      : AttrItem -> [ Ref(CDATAItem) ]
psv_attr_uri           : AttrItem -> uriReference
psv_attr_local_name    : AttrItem -> string
psv_attr_owner_elem    : AttrItem -> Ref(ElemItem)

/* Infoset item for schema type */
psv_attr_schema_type   : AttrItem -> Ref(ElemItem)

/* peripheral */
psv_attr_type          : AttrItem -> string
psv_attr_spec_flag     : AttrItem -> SpecifiedFlag
psv_attr_default       : AttrItem -> [ Ref(CDATAItem) ]
psv_attr_start_entity  : AttrItem -> Ref(EntityMarker)
psv_attr_end_entity    : AttrItem -> Ref(EntityMarker)
```

The function *dm_attrNode* maps an *AttrItem* to an *AttrNode*:

```
dm_attrNode : AttrItem -> AttrNode
fun dm_attrNode(a) {
   name = qnameValue(psv_attr_uri a, psv_attr_local_name a, ref(Def_QName))
   /* An attribute value should be one string */
   value = head(dm_collapse_strings(map_list dm_node (psv_attr_children a)))
   type = dm_schema_type(psv_attr_schema_type a)
   /* convert value to appropriate type */
   return attrNode(name, (dm_coerce_string type value))
}
```

A.6 Namespace Items

The following accessors are defined for namespaces:

```
/* core */
psv_ns_prefix         : NSItem -> string
psv_ns_uri            : NSItem -> uriReference
psv_ns_value          : NSItem -> [ Ref(CDATAItem) | Ref(EntityMarker) ]
psv_ns_owner_elem     : NSItem -> Ref(ElemItem)
```

The function *dm_nsNode* maps an *NSItem* to an *NSNode*:

```
dm_nsNode : NSItem -> NSNode
fun dm_nsNode(i) {
   /* what happens to NS node's CDATA children? */
```

```
    return nsNode(stringValue(psv_ns_prefix i, ref(Def_string), []),
                  urirefValue(psv_ns_uri i, ref(Def_uriReference)))
}
```

A.7 Processing Instruction Items

The following accessors are defined on processing-instruction information items:

```
/* core */
psv_pi_target         : PIItem -> string
psv_pi_value          : PIItem -> string
psv_pi_base_uri       : PIItem -> uriReference
psv_pi_parent         : PIItem
                      -> (Ref(ElemItem) | Ref(DocItem) | Ref(DocTypeItem))
```

The function *dm_piNode* maps an *PIItem* to an *PINode*:

```
dm_piNode : PIItem -> PINode
fun dm_piNode(i) {
  return piNode(stringValue(psv_pi_target i, ref(Def_NCName), []),
                stringValue(psv_pi_value i, ref(Def_string), []))
}
```

A.8 Comment Items

The following accessor is defined on comment information items:

```
/* core */
psv_comm_value         : CommentItem -> string
psv_comm_parent        : CommentItem
                       -> (Ref(ElemItem) | Ref(DocItem) | Ref(DocTypeItem))
```

The function *dm_commNode* maps an *CommentItem* to an *CommentNode*:

```
dm_commNode : CommentItem -> CommentNode
fun dm_commNode(i) {
  return  commNode(stringValue(psv_comm_value i, ref(Def_string), []))
}
```

A.9 Other Information Items

The data model represents document type, skipped entity reference, entity markers, and character data markers as instances of the opaque data type *InfoItemNode*. For completeness, the following accessors are defined for these information items.

```
/* Skip Entity Item : core */
psv_skip_entity_item : SkipEntityItem -> string
psv_skip_entity_ref   : SkipEntityItem -> Ref(EntityItem)
psv_skip_entity_parent : SkipEntityItem -> Ref(ElemItem)

/* CDATA Item : core */
psv_cdata_code        : CDATAItem -> Code
psv_cdata_parent      : CDATAItem -> Ref(ElemItem)
/* CDATA Item : peripheral */
psv_cdata_whitespace : CDATAItem -> boolean
```

```
psv_cdata_predefined : CDATAItem -> boolean

/* Document Type Item :  peripheral */
psv_doctype_entity   : DocTypeItem -> Ref(EntityItem)
psv_doctype_children : DocTypeItem -> [ Ref(CommentItem) | Ref(PIItem) ]
psv_doctype_parent   : DocTypeItem -> Ref(DocItem)

/* Entity Item : core */
psv_entity_name      : EntityItem -> string
psv_entity_sysid     : EntityItem -> string
psv_entity_pubid     : EntityItem -> string
psv_entity_notat     : EntityItem -> Ref(NotationItem)
psv_entity_base_uri  : EntityItem -> uriReference

/* Entity Item : peripheral */
psv_entity_type      : EntityItem -> string
psv_entity_content   : EntityItem -> string
psv_entity_encoding  : EntityItem -> string
psv_entity_standalone: EntityItem -> string /* "yes", "no", "notpresent" */

/* Entity Marker : core */
psv_entity_marker_entity : EntityMarker -> Ref(EntityItem)
psv_entity_marker_parent : DocTypeItem
                         -> (Ref(ElemItem) | Ref(AttrItem) | Ref(NSItem))

/* Notation Item : core */
psv_notation_name    : NotationItem -> string
psv_notation_sysid   : NotationItem -> string
psv_notation_pubid   : NotationItem -> string
psv_notation_base_uri: NotationItem -> uriReference

/* CDATA Marker : core */
psv_cdata_marker_parent : CDataMarker
                        -> (Ref(ElemItem) | Ref(AttrItem) | Ref(NSItem))
```

B Candidate Operators for XML Query Algebra

There is a close correspondence between the data model and the algebraic operations that are applied to instances of the data model. In this appendix, we identify those algebraic operators that are candidate operators for the algebra.

B.1 Equality Operators

An algebra typically defines at least one equality operator for all types in the data model. We assume the algebra includes two equality operators: *node_equal* and *ref_equal*, which define equality on node values and on node references. We assume that these operators are provided by the data-model implementation and conform to the constraints specified in Constraints on the Data Model. The algebra may define other equality operators, for example, that perform coercions between values of different types.

B.2 Serialization Operators

In the XPath data model, every kind of node has an associated string value, i.e., a string serialization of the node's value. The data model already provides an operator for serializing a *ValueNode* as a string. The algebra also may include one or more operators to serialize a *Node* as a string:

```
serialize : Node -> StringValue
```

One serialization operator may emit a node value in Canonical XML. Other operators may choose other serializations.

B.3 Order Operators

Given an instance of the data model, an ordering relation on its nodes can be defined. For example, document order, i.e., the absolute order of nodes in an input document, is a property that a user of the XML Query Language may want to query.

Given a single input document, a global, total order can be defined on nodes. In general, however, it may not be possible to define a global order, for example when a data model instance contains nodes from multiple input documents. Therefore, it may be useful and necessary to have multiple ordering relations, e.g., one that is global and one that is partial:

```
globalOrder  : (Instance, Node) -> IntegerValue
partialOrder : (Instance, Node) -> IntegerValue
```

Order operators must be defined over a particular instance of the data model. The functions return an integer order value given an instance of the data model and a node contained in that instance.

B.4 Accessors for Reference Values

Although reference values are often contained in attribute nodes, they typically represent element-to-element relationships. For convenience, the algebra may define additional accessor functions that produce the references associated with a particular element.

For example, the *idrefs* function takes an *ElemNode* and returns a list of references to element nodes that are referenced by IDREF values in the element's attribute and children components; this accessor is defined in terms of existing accessors:

```
idrefs : ElemNode -> [ Ref(ElemNode) ]
fun idrefs(n) {
  concat_list(
    flat_list [ referent(v) | a <- list(attributes(n)), v = value(a),
                isIDREFValue(v) ],
    flat_list [ referent(c) | c <- children(n), isIDREFValue(c) ])
   )
}
```

The function is defined using a *list comprehension*, which is based on a familiar notation for sets. The second expression above:

```
[ referent(c) | c <- children(n), isIDREFValue(c) ]
```

is read as follows: return the list of the referent values of each node *c* such that *c* is a child of *n* and *c* is a *IDREFValue*. Since *referent* produces a list, the result of this expression is a list of lists: *flat_list* takes a list of lists and returns the flattened list, and *concat_list* concatenates the two lists into one. The *attributes* accessor returns a set, not a list, so the *list* function converts

(non-deterministically) a set to a list.

The *links* accessor is analogous to *idrefs*, but returns non-null URI values that occur in the element's attributes and children components; it is defined as:

```
links : ElemNode -> [ Ref(ElemNode) ]
fun links(n) {
  concat_list(
    flat_list [ referent(v) | a <- list(attributes(n)), v = value(a),
                isURIRefValue(v), not(referent(v) = NaR) ],
    flat_list [ referent(v) | c <- children(n), isURIRefValue(c),
                not(referent(c) = NaR)])
}
```