XQuery 1.0 Formal Semantics

W3C Working Draft 26 March 2002

This version:

http://www.w3.org/TR/2002/WD-query-semantics-20020326/

Latest version:

http://www.w3.org/TR/query-semantics/

Previous versions:

http://www.w3.org/TR/2001/WD-query-semantics-20010607/
http://www.w3.org/TR/2001/WD-query-algebra-20010215/
http://www.w3.org/TR/2000/WD-query-algebra-20001204/

Editors:

Denise Draper (XML Query WG), Nimble Technology <ddraper@nimble.com>

Peter Fankhauser (XML Query WG), Infonyte GmbH <fankhaus@infonyte.com>

Mary Fernández (XML Query WG), AT&T Labs - Research <mff@research.att.com>

Ashok Malhotra (XML Query and XSL WGs), Microsoft <ashokma@microsoft.com>

Kristoffer Rose (XSL WG), IBM Research <krisrose@us.ibm.com>

Michael Rys (XML Query WG), Microsoft <mrys@microsoft.com>

Jérôme Siméon (XML Query WG), Lucent Technologies <simeon@research.bell-labs.com>

Philip Wadler (XML Query WG), Avaya <wadler@avaya.com>

Abstract

This document presents the formal semantics of XQuery [XQuery 1.0: A Query Language for XML].

Status of this Document

This is a public W3C Working Draft for review by W3C Members and other interested parties. This section describes the status of this document at the time of its publication. It is a draft document and may be updated, replaced, or obsoleted 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." A list of current public W3C technical

reports can be found at http://www.w3.org/TR/.

Much of this document is the result of joint work by the XML Query and XSL Working Groups, which are jointly responsible for XPath 2.0, a language derived from both XPath 1.0 and XQuery. This document defines the formal semantics for XQuery 1.0. A future version of this document will also define the formal semantics for XPath 2.0. Further information on the progress of the work of the XML Query WG can be found at http://www.w3.org/XML/Query.

This document is a work in progress. It contains many open issues, and should not be considered to be fully stable. Vendors who wish to create preview implementations based on this document do so at their own risk. While this document reflects the general consensus of the working groups, there are still controversial areas that may be subject to change.

A complete list of issues is maintained in **[C Issues]**. In particular, the reader should note the following important issues.

- The XQuery type system is based on XML Schema, but some misalignments exist. See [**Issue-0104: Support for named typing**].
- The semantics of sortby and constructor expressions is still under discussion. See [**Issue-0109: Semantics of sortby**] and [**Issue-0110: Semantics of element and attribute constructors**].
- The current semantics does not completely cover all functions. Some functions used in the Formal semantics, or some functions from the [XQuery 1.0 and XPath 2.0 Functions and Operators] document need additional semantics specification. See [**Issue-0135: Semantics of special functions**].

Comments on this document should be sent to the W3C XML Query mailing list www-xml-query-comments@w3.org; note that any email information sent to this list is publicly available in the W3C XML Query mailing list archive http://lists.w3.org/Archives/Public/www-xml-query-comments/.

XQuery 1.0 has been defined jointly by the XML Query Working Group (part of the XML Activity) and the XSL Working Group (part of the Style Activity).

Table of Contents

Appendices

---

1 Introduction

This document defines the formal semantics of XQuery. The present document is part of a set of documents that together define the XQuery 1.0 language:

- [XQuery 1.0: A Query Language for XML] introduces the XQuery 1.0 language, defines its capabilities from a user-centric view, and defines the language syntax.
- [XQuery 1.0 and XPath 2.0 Data Model] formally specifies the data model used by XQuery to represent the content of XML documents. The XQuery language is formally defined by operations on this data model.
- [XQuery 1.0 and XPath 2.0 Functions and Operators] lists the functions and operators defined for the XQuery language and specifies to which arguments they can be applied and what the result should be.

The scope and goals for the XQuery language were laid out in the charter of the XML Query Working Group and in the XQuery requirements [XML Query 1.0 Requirements].

XQuery is a powerful language, capable of selecting and extracting complex patterns from XML documents and of reformulating them into results in arbitrary ways. This document defines the semantics of XQuery by giving a precise formal meaning to each of the constructions of the XQuery specification in terms of the XQuery data model. This document assumes that the reader is already familiar with the XQuery language.

Two important design aspects of XQuery is that it is *functional* and *typed*. These two aspects play an important role in the XQuery Formal Semantics.

XQuery is a functional language. It is built from expressions , called *queries*, rather than statements. Every construct in the language (except for the query prolog) is an expression and expressions can be composed arbitrarily. The result of one expression can be used as the input to any other expression, as long as the type of the result of the former expression is compatible with the input type of the later expression with which it is composed. Another aspect of the functional approach is that variables are always passed by value and their value cannot be modified through side-effects.

XQuery is a typed language. Types can be imported from one or several XML Schemas (typically describing the documents that will be processed), and the XQuery language can then perform operations based on these types (e.g., using a `treat as` operation). In addition, XQuery also supports a level of *static type analysis*. This means that the system can perform some inference on the type of a query, based of the type of its inputs. Static typing allows early error detection, and may be the basis for certain forms of optimization. The type system of XQuery is based on [XML Schema Part 1], but does not support all the features of XML Schema. For example, XQuery does not provide support for derivation. The relationship between the XQuery type system and XML Schema is defined in **[3.5 Importing types from XML Schema]**. Issues with respect to the XQuery type system are discussed in **[3.6 Major type issues]**.

The XQuery formal semantics builds on long-standing traditions in the database and in the programming

languages communities. In particular, we have been inspired by works on SQL [SQL], OQL [ODMG], [BKD90], [CM93], and nested relational algebra (NRA) [BNTW95], [Col90], [LW97], [LMW96]. We have also been inspired by systems such as Quilt [Quilt], UnQL [BFS00], XDuce [HP2000], XML-QL [XMLQL99], XPath [XPath], XQL [XQL99], XSLT [XSLT 99], and YaTL [YAT99]. Additional citations are found in the bibliography **[B References]**.

This document is organized as follows. In **[2 Preliminaries]**, we introduce concepts and notations used for defining the XQuery formal semantics. In **[3 The XQuery Type System]**, we describe the XQuery type system, operations on the XQuery type system, and explain the relationship between the XQuery type system and XML Schema. In **[4 Semantics of Expressions]**, we describe the dynamic and static semantics of XQuery. In **[5 The Query Prolog]**, we describe the semantics of the XQuery prolog. In **[6 Additional Semantics of Functions]**, we describe any additional semantics required for functions.

2 Preliminaries

> **Ed. Note: Status:** This section contains introductory material and is mostly new.

In this section, we provide background concepts for the XQuery formal semantics and introduce the notations that are used.

2.1 Processing model

We first define a processing model for query evaluation. This processing model is not intended to be a model for an actual implementation, although a (naive) implementation might be obtained from it. It does not require or constrain any implementation technique, but any implementation should produce the same results as the one obtained by following this processing model and applying the rest of the formal semantics specification.

The processing model is based of four phases; each phase consumes the result of the previous phase and generates output for the next phase:

1. **Parsing.** The grammar for the XQuery syntax is defined in [XQuery 1.0: A Query Language for XML]. Parsing may generate syntax errors. If no error occurs, some internal representation of the parsed syntax tree is created.

2. **Normalization.** To simplify the semantics specification, we first perform some normalization over the query. The XQuery language provides many powerful features that make querys simpler to write and use, but are also redundant. For instance, a complex `for` expression might be rewritten as a composition of several simple `for` expressions. The language composed of these simpler query is called the XQuery *Core language* and is a subset of the complete XQuery language. The grammar of the XQuery Core language is given in **[A Normalized core grammar]**.

   During the normalization phase, each XQuery query is mapped into its equivalent query in the core. (Note that this has nothing to do with Unicode Normalization which works on character strings). Normalization works by bottom-up application of normalization rules over expressions, starting with normalization of literals. After normalization, the full semantics can be obtained just by giving a semantics to the normalized Core query. This is done during the last two phases.

   > **Ed. Note:** The query prolog needs to be processed before normalization starts. **[5 The Query Prolog]** explains how the query prolog is processed. This is not yet reflected in the

processing model. See [Issue-0130: When to process the query prolog].

3. **Static type analysis.** Static type analysis checks if each query is type safe, and if so, determines their static types. Static type analysis is defined only for Core query. Static type analysis works by bottom-up application of type inference rules over expressions, taking the type of literals and the known types of input documents into account.

   Static type analysis can result in a *static error*, if the query is not type-safe. For instance, a comparison between an integer value and a string value might be detected as an error during the static type analysis. If static type analysis succeeds, it results in a syntax tree where each sub-expression is "decorated" with its static type, and in an *output type* for the result of the query.

   Static typing does not imply that the content of XML documents must be rigidly fixed or even known in advance. The XQuery type system accommodates "flexible" types, such as elements that can contain any content. Schema-less documents are handled in XQuery by associating a standard type with the document, such that it may include any legal XML content.

4. **Dynamic Evaluation.** This is the phase in which the result of the query is computed. The semantics of evaluation is defined only for Core query terms. Evaluation works by bottom-up application of evaluation rules over expressions, starting with evaluation of literals. This guarantees that every expression can be unambiguously reduced to a value, which is the final result of the query.

The first three phases above are "analysis-time" (sometimes also called "compile-time") steps, which is to say that they can be performed on a query before examining any input document. Indeed, analysis-time processing can be performed before such document even exists. Analysis-time processing can detect many errors early-on, e.g., syntax errors or type errors. If no error occurs, the result of analysis-time processing could be some compiled form of query, suitable for execution by a compiled-query processor. The last phase is an "execution-time" (sometimes also called "run-time") step, which is to say that it is performed on the actual input document.

Static analysis catches only certain classes of errors. For instance, it can detect a comparison operation applied between incompatible types (e.g., `xsd:int` and `xsd:date`). Some other classes of errors cannot be detected by the static analysis and are only detected at execution time. For instance, whether an arithmetic expression on 32 bits integers (`xsd:int`) yields an out-of-bound value can only be detected by looking at the data, and is raised as an evaluation error.

While implementations may be free to employ different processing models, the XQuery static semantics relies on the existence of a static type analysis phase that precedes any access to the input data. Statically typed implementations are required to find and report static type errors, as specified in this document. It is still an open issue (See [Issue-0098: Implementation of and conformance levels for static type checking]) whether to make the static type analysis phase optional to allow implementations to return a result for a type-invalid query in special cases where the query happens to perform correctly on a particular instance of input data. (Note that this is not the same as merely permitting that the static type error is emitted at evaluation time as we do below.)

Notice that the separation of logical processing into phases is not meant to imply that implementations must separate analysis-time from evaluation-time processing: query processors may choose to perform all phases simultaneously at evaluation-time and may even mix the phases in their internal implementations. All the processing model defines is what the final result of processing should be.

The above processing phases are all internal to the query processor. They do not deal with how the query processor interacts with the outside world, notably how it accesses actual documents and types. A typical

query engine would support at least three other important processing phases:

1. **XML Schema import phase.** The XQuery type system is based on XML Schema. In order to perform static type analysis, the XQuery processor needs to build type descriptions that correspond to the schema of the input documents. This phase is achieved by mapping all schemas required by the query into the XQuery type system. The XML Schema import phase is described in this document in **[3.5 Importing types from XML Schema]**.

2. **XML loading phase.** During the evaluation phase, expressions are applied on instances of the [XQuery 1.0 and XPath 2.0 Data Model]. How XML documents are loaded into the data model is described in the [XQuery 1.0 and XPath 2.0 Data Model] and is not be discussed further here.

3. **Serialization phase.** Once the query is evaluated, processors might want to serialize the result of the query as actual XML documents. Serialization of data model instances is still an open issue (See [**Issue-0116: Serialization**]) and is not be discussed further here.

We do not describe the parsing phase formally. Notably, we do not specify data structures for the syntax trees. Instead, we use the XQuery concrete syntax directly as a support for the formal semantics specification. More details about parsing can be found in the [XQuery 1.0: A Query Language for XML] document. No further discussion of parsing is included here.

For the other three phases (normalization, static type analysis and evaluation), instead or giving an algorithm in pseudo-code, we describe the semantics formally by means of *inference rules*. Inference rules provide a notation to describe how the semantics of an expression derives from the semantics of its sub-expressions. Hence, they provide a concise and precise way for specifying bottom-up algorithms, as required by the normalization, static type analysis, and evaluation phases.

2.2 Data Model

In this section we first present the basic components of the XQuery data model pertinent to the semantic description. The remainder of the section is devoted to several related features that merit special attention: *node identity*, *order*, and *error* values.

2.2.1 Data model components

The components manipulated in XQuery, such as *simple-typed values* and *nodes* (attributes, elements, etc.), are defined in the [XQuery 1.0 and XPath 2.0 Data Model] document. In this section we present only a short review of these components. We refer the reader to the [XQuery 1.0 and XPath 2.0 Data Model] document for more details.

There are five kinds of components in the [XQuery 1.0 and XPath 2.0 Data Model]: Error, SimpleValue, Node, Sequence, and SchemaComponent.

- A single error value.
- A SimpleValue: a value from any of the value spaces of XML Schema simple types, as defined in [XML Schema Part 2].
- A Node; which is one of the following kinds: DocumentNode, ElementNode, AttributeNode, NamespaceNode, CommentNode, PINode and TextNode. XML documents and their content are

represented in the data model as trees composed of nodes and values. [XQuery 1.0 and XPath 2.0 Data Model] defines a mapping from the PSVI (Post Schema Validation Information Set) to such trees.

- A Sequence: which is an ordered collection of nodes, an ordered collection of simple-typed values, or an ordered collection of *items* (i.e., any mixture of nodes and simple-typed values). An important characteristic of sequences is that they cannot be nested. That is, all sequences are "flat", and cannot be composed of other sequences. Moreover, there is no distinction between a single item and a singleton sequence containing that item. That is, a single item and a singleton sequence containing that item are equivalent. It is still open whether sequences of entire documents are permitted **[Issue-0068: Document Collections]**.
- A SchemaComponent: the type of a node. The content and use of this schema component is still an open issue.

[XQuery 1.0 and XPath 2.0 Data Model] defines constructors, functions to construct each of these kinds of data model component, as well as accessors, functions to access their contents. For instance, the xf:children function can be used to retrieve the children nodes of an element node.

2.2.2 Node identity

In the [XQuery 1.0 and XPath 2.0 Data Model], nodes have an *identity*. Node identity follows from the unique location of any node within exactly one XML document (or document fragment, in the case of XML being constructed within the query itself). It is possible for two expressions to return not only the same value, but also the exact same node. On the other hand, two expressions could have results with the same document structure, but different identities. XQuery supports comparison of two nodes using either "node equality" (equality by identity) or "value equality" (equality by equal values), for instance using one of the two operators == or =.

Simple-typed values do not have an associated identity and are therefore always compared by value equality.

The difference between node equality and value equality is illustrated on the following example:

```
let $a1 := <book><author>Suciu</author></book>,
    $a2 := <author>Suciu</author>,
    $a3 := $a1/author
return
    ($a1/author == $a3), {-- true, they refer to the same node --}
    ($a1/author = $a2),  {-- true, they have the same value --}
    ($a1/author == $a2)  {-- false, they are not the same node --}
```

All XQuery's operators preserve node identity with the exception of explicit copy operations and node (element, attribute, etc.) constructors. An element constructor always creates a deep copy of its attributes and children nodes. Other operators, such as sequence construction or path expressions, do not result in copies of the corresponding nodes. So, for example, `($a3, $a2)` creates a new sequence of nodes, the first of which has the same identity as `$a1/author`.

2.2.3 Document order and sequence order

There are two kinds of order in XQuery: sequence order and document order.

Sequence order. Sequences of items in the XQuery data model are ordered. Sequence order refers to the order

of these items within a given sequence.

Document order. Document order refers to the total order among all nodes within a given document. It is defined as the order of appearance of the nodes when performing a pre-order, depth-first traversal of a tree. For elements, this corresponds to the order of appearance of their opening tags in the corresponding XML serialization. Document order is equivalent to the definition used in [XML Path Language (XPath) : Version 1.0].

Depending on the context, it may be possible to talk about two different notions of order for the same (set of) nodes. For example:

```
let $e := <list>
              <item id="1"/>
              <item id="2"/>
          </list>,
    $s := ($e/item[@id='2'], $e/item[@id='1'])
...
```

The item "1" is before item "2" in document order, but the same two nodes are in the opposite order in the sequence `$s`.

In the case of nodes among several documents, the order between the document is implementation defined but must be stable. I.e., it must not change for the duration of query evaluation. Support for document order among elements in distinct documents or document fragments is discussed in [**Issue-0003: Document Order**].

2.2.4 Errors

Some queries result in an error. Errors that are detected at compile-time, like parse errors or type errors, are reported to the user by the system. Dynamic errors must return the Error value defined by the [XQuery 1.0 and XPath 2.0 Data Model].

Whenever necessary, we use the notation `dm:error()` to refer to the error value in the [XQuery 1.0 and XPath 2.0 Data Model]. General handling of errors in XQuery is still an open issue (See [**Issue-0094: Static type errors and warnings**] and Issue-98 in [XQuery 1.0: A Query Language for XML]).

2.3 Schemas and types

The Data Model establishes the space of values that may be manipulated by XQuery, but it does not establish a complete type system. For example, we may know that a node is an ElementNode, but that does not tell us what kind of element it is, what its legal contents is, etc.

The XQuery type system is based on [XML Schema Part 1]. An introduction to XML Schema can be found in the primer [XML Schema Part 0]. An important notion underlying XML Schema are regular expressions, and their tree counterparts, regular tree grammars. An introduction to regular (tree) languages can be found in [Languages] or in [TATA]. In order to denote regular expressions, the XQuery type system uses the familiar notation of XML DTDs. We refer the reader to [XML] for more on DTDs.

> **Ed. Note:** We do not give an introduction to regular expressions and tree grammars for now, but a future version of the document should add some introductory text about: regular expressions,

tree grammars, and basic algorithms over these that are relevant for XQuery. Notably, some text about closure properties, complexity, subsumption algorithm, etc. could be provided there.

This section presents an introduction to (static) type systems in general, and a conceptual overview of the XQuery type system in particular. The XQuery type system is discussed formally in **[3 The XQuery Type System]**.

2.3.1 The elements of a (static) type system

A type system relates values, types, and expressions. A (static) type system requires several constituent parts:

- **A universe of possible types**. Every type system starts with a set of *primitive types* and most (including XQuery's) have *type constructors* that allow the construction of *complex types*.

  In XQuery, the primitive types are the *built-in simple datatypes* types of [XML Schema Part 2]. There are also type constructors for sequences and node types such as attributes and elements. These correspond to the concepts of group, attribute and element in [XML Schema Part 1] (or more precisely, in the formalization of XML Schema in [XML Schema : Formal Description]).

- **A notation for types.** A syntax is necessary to define and manipulate types. In order to write the type related semantics of XQuery, this document uses its own notation, which is used notably in type inference and dynamic semantics rules.

- **A notion of instances, and domain of a type.** Formally, a type is defined to be the (usually infinite) set of instance values that belong to that type. Thus the type xs:integer consists of the set of all integer values, and the type `element address { xs:string }` consists of all elements named "address" whose contents are a single string value.

- **A notion of subtyping.** Subtyping is a relationship between type, i.e., it relates types to each other.

  Subtyping plays a crucial role in a type system. Most notably, it is used to determine the legality of arguments to functions.

- **Rules that relate expressions to types.** The heart of static type inference is associating a *static type* with each expression in a query. In a strongly-typed language like XQuery, the static type is a *guarantee* that, no matter what the input to the query is, the result of evaluating the expression is a value that belongs to its static type.

- **Rules that relate values to types.** During query evaluation, expressions are evaluated to yield values. The dynamic component of a typing system determines the actual types of data values as they are computed.

2.3.2 Subtyping

One type is a subtype of another if every value belonging to the first type also belongs to the other. Here is a simple example of subtyping:

```
( xs:double )*
( xs:integer | xs:double )*
```

The first type is a subtype of the second, because any sequence that contains only doubles is also an example of a sequence containing either integers or doubles.

We write $Type_1$ `<:` $Type_2$ to indicate that type $Type_1$ is a subtype $Type_2$. Note that we use the subtype notation `<:` when defining the XQuery semantics, but it is *not* part of the XQuery syntax.

2.3.2.1 Type equivalence

Two types are *equivalent* if every value of one type is also a value of the other type, and vice versa, i.e., two types are equivalent if and only if they are both subtypes of the other. Here is a simple example of two equivalent types:

```
( xs:integer | xs:double )*
( xs:double | xs:integer )*
```

Both of these types mean a sequence of any number of doubles or integers.

We write type equivalence as $Type_1$ = $Type_2$, where $Type_1$ and $Type_2$ are both types. If $Type_1$ = $Type_2$ then $Type_1$ `<:` $Type_2$ and $Type_2$ `<:` $Type_1$ are true.

2.3.2.2 Type substitutability

Subtyping plays a crucial role in the semantics of function application since all functions have an associated signature that declares the types of input parameters and the type of the result of the function.

Intuitively, one can call a function not only by passing an argument of the declared type, but also any argument whose type is a subtype of the declared type. This property is called *type substitutability*.

> **Ed. Note:** This section should contain an example.

2.3.2.3 Subtyping and XML Schema derivation

Subtyping in XQuery is not the same as "derivation" in XML Schema. A derived type is always a subtype of its base type, but the opposite is not true. For instance, consider the following types:

```
( xs:double )+
xs:double, ( xs:integer | xs:double )*
```

The first type is a subtype of the second, because any sequence that contains one or more doubles is also an example of a sequence containing a double followed by a sequence of zero or more integers or doubles. But derivation as defined in XML Schema does not hold between these two types.

The relationship between subtyping and XML Schema derivation is discussed in **[Issue-0019: Support derived types]**.

2.3.3 Static types and dynamic types

One consequence of having a static type system is that for the same expression, a type is first computed during the static phase, then a type for the actual *value* which is the result of that expression is computed during the dynamic phase. We use the terms static type and dynamic type respectively for each of these computed types for a given expression. It is important to note that for the same expression the dynamic type is often more precise than the static one, since the real value is not known statically. Indeed, the static type can be considered as an *approximation* of the dynamic one based on the available static information. Consider, for example, an XPath expression

```
$x/(foo | bar)
```

In this expression the static type might be

```
( element foo | element bar )*
```

If during execution, `$x` happens to contain exactly two `foo` elements, the dynamic type of (the result of evaluating) `$x` is a sequence of two `foo` elements.

```
( element foo, element foo ).
```

Of course, the basic guarantee of static type inference is that the dynamic type is always a subtype of the static type. Note that dynamic types are *specific*: they never contain choices, all groups, or quantifiers.

> **Ed. Note:** MFF: This next paragraph should be changed to describe type annotations.

The XQuery data model also defines SchemaComponents associated with Nodes. SchemaComponents relate to the type associated with a Node through schema validation. This may be different from either the static or the dynamic type of the node. The relationship between static types, dynamic types and schema components in the data model is still an open issue. See **[Issue-0018: Align algebra types with schema]**.

2.4 Functions

2.4.1 Functions and operators

The [XQuery 1.0 and XPath 2.0 Functions and Operators] document provides a number of useful basic functions over the components of the XQuery data model (simple-typed values, nodes, sequences, etc). We use a number of these functions in the course of describing the XQuery semantics. Here is the list of functions from the [XQuery 1.0 and XPath 2.0 Functions and Operators] document that are used in the XQuery formal semantics:

- `op:intersect`
- `op:union`
- `op:except`
- `op:concatenate`
- `op:boolean-and`
- `op:boolean-or`
- `op:union-value`
- `op:except-value`
- `op:to`
- `op:numeric-equal`
- `op:numeric-add`

- op:get-end-datetime
- op:numeric-subtract
- op:get-duration
- op:get-start-datetime
- op:numeric-multiply
- op:numeric-divide
- op:numeric-mod
- op:boolean-equal
- op:datetime-equal
- op:duration-equal
- op:hex-binary-equal
- op:base64-binary-equal
- op:numeric-greater-than
- op:datetime-greater-than
- op:duration-greater-than
- op:numeric-less-than
- op:datetime-less-than
- op:duration-less-than
- op:node-equal
- op:node-before
- op:node-after
- op:numeric-unary-plus
- op:numeric-unary-minus
- op:operation
- xf:false
- xf:true
- xf:length
- xf:boolean
- xf:item-at
- xf:get-namespace-uri
- xf:get-local-name
- xf:round
- xf:compare
- xf:not
- xf:not3
- xf:empty-sequence
- xf:root
- xf:index-of

2.4.2 Functions and static typing

Many functions in the [XQuery 1.0 and XPath 2.0 Functions and Operators] document are *generic*: they perform operations on arbitrary components of the data model, e.g., any kind of node, or any sequence of items. For instance, the `xf:distinct-nodes` function removes duplicates in any sequence of nodes. As a result, the signature given in the [XQuery 1.0 and XPath 2.0 Functions and Operators] document is also generic. For instance, the signature of the `xf:distinct-nodes` function is:

```
define function xf:distinct-nodes(node*) returns node*
```

If applied as is, this signature would result in poor static type information. In general, it would be preferable if some of these function signatures could take the type of input parameters into account. For instance, if the function `xf:distinct-nodes` is applied on a parameter of type `element a*, element b`, one can easily deduce that the resulting sequence is a collection of either `a` or `b` elements.

In order to provide better static typing, we specify typing rules for some of the functions in [XQuery 1.0 and XPath 2.0 Functions and Operators]. These additional typing rules are given in **[6 Additional Semantics of Functions]**. Here is the list of functions that are given specific typing rules.

- `dm:error`
- `op:union`
- `op:intersect`
- `op:expect`
- `op:to`

Ed. Note: This list reflects the content of Sections 6, but should be expanded. See **[Issue-0135: Semantics of special functions]**.

2.4.3 Data Model Accessors and XPath Axes

The XQuery Data Model provides operations to construct or access component of the Data Model. Some of these operations are used in the course of defining the formal semantics. We use the namespace prefix `dm:` those functions to distinguish them for other functions from the [XQuery 1.0 and XPath 2.0 Functions and Operators] document.

Here is a list of data model constructors or accessors used for formal semantics specification.

- `dm:error`
- `dm:atomic-value`
- `dm:children`
- `dm:attributes`
- `dm:parent`
- `dm:name`
- `dm:node-kind`
- `dm:dereference`
- `dm:empty-sequence`
- `dm:namespaces`
- `dm:type`

- `dm:typed-value`
- `dm:attribute-simple-node`
- `dm:element-simple-node`
- `dm:text-node`

XPath axes are used to describe tree navigation in instances of the XQuery data model. The semantics of axis navigation is described in terms of data model functions. Some of the XPath axes are directly supported through existing data model accessors. Some axes (e.g., ancestor) are defined as a simple recursive application of a data model accessor and others (e.g., following) require more complex computation on top of existing data model accessors. The correspondence between XPath axis and data model accessors is specified in section **[4.3.1 Axis Steps]**

2.4.4 Other formal semantics functions

In a few cases, the formal semantics makes use of some functions that are not currently in the [XQuery 1.0 and XPath 2.0 Functions and Operators] or in the [XQuery 1.0 and XPath 2.0 Data Model] documents. We use the namespace prefix `fs:` for those functions, to distinguish them from functions in the [XQuery 1.0 and XPath 2.0 Functions and Operators] document and from Data Model accessors.

Here is a list of additional functions used for formal semantics specification.

- `fs:document`
- `fs:distinct-doc-order`
- `fs:data`
- `fs:unique-item-at`

2.5 Notation

> **Ed. Note: Status:** This section is intended to provide an introduction to formal notations, such as inference rules, for readers which are not familiar with these notations. It can be skipped by the reader which is familiar with these notations. This introductory material is still considered a draft and will be improved in further rewritings of the Formal Semantics document.

In order to make the semantics specification both concise and precise, we use logical notation in the form of logical inference rules. In this section, we introduce the notations for inference rules and sorts of semantic values, specifically environments.

2.5.1 Judgments and patterns

> **Ed. Note:** The following is a somewhat terse explanation of inference rules using formal semantic lingo informally. It needs some examples.

The basic building block of an inference rule is a *judgment* that expresses some property. A judgment is specified by a description that contains a combination of basic *patterns*, each identifying a sort of value that can be inserted, interspersed with symbols. Symbols must occur literally in judgments between values of a sort that corresponds to the used patterns.

A pattern is identified by one or more italic words. The names of patterns are significant: each pattern name corresponds to a sort of value that can be substituted legally for the pattern. For this reason, a pattern is

sometimes called a *metavariable* that is *instantiated* to some value of the appropriate sort.

We employ the following conventions for pattern names:

1. patterns for syntactic sorts denoting *lexical items* begin with upper case and their names correspond to terminals or nonterminals in the XQuery grammar. For example, *Expr* is a pattern that can be instantiated with any XQuery *Expr*.
2. patterns for semantic sorts, to be defined, begin with lower case letters. For example, denotes a semantic value.

Furthermore, all patterns may appear with subscripts (e.g. $Expr_1$, $Expr_2$), which simply name different instances of the same pattern sort.

For example, '3 => 3' and '$x+0 => 3' are both instances of the judgment described by '*Expr* => *value*'. It is no problem that we are *overloading* the symbol '3' to denote both the '3' in an expression and the value 3: this is unambiguous because of the strict use of sorts in the judgment descriptions.

Finally, we require that each distinct pattern is instantiated to a single expression or value so if the same pattern occurs twice in a judgment description then it should be instantiated with the same value. This means that '3 => 3' is an instance of the judgment described by '*Expr* => *Expr*' but '$x+0 => 3' is not. (Both, however, are instances of the judgment described by '$Expr_1$ => $Expr_2$'.)

2.5.2 Inference rules

Inference rules are used to express the logical relation between judgments and define how complex judgments can be concluded from simpler premise judgments. (As explained in **[2.1 Processing model]**, this approach lends itself well to bottom-up algorithms.)

A logical inference rule is written as two collections of *premise* and *conclusion* judgments, written above and below a dividing line, respectively:

$$\frac{premise_1 \ldots premise_n}{conclusion_1 \ldots conclusion_n}$$

The interpretation of an inference rule is: if all the premise judgments above the line are true, then we know that each of the conclusion judgments below the line are also true.

Here is a simple example to illustrate (based on the example judgment from above described by '$Expr_1$ => $Expr_2$'):

$$\frac{\$x \Rightarrow 0 \qquad 3 \Rightarrow 3}{\$x + 3 \Rightarrow 3}$$

This rule expresses the property of the evaluation judgment '*Expr*=>*value*': if an expression containing the variable reference '$x' yields the value zero, and the literal expression '3' yields the value '3', then we know that '$x + 3' yields the value '3'.

It is also possible for the expression above the line to have no judgments at all; this simply means that the expression below the line is true under all premises:

$$\frac{}{3 \Rightarrow 3}$$

This judgment says that evaluating the expression '3' always yields the value three.

The rules above are expressed in terms of specific variables and values, but usually rules are more abstract. That is, the judgments they relate contain patterns that match sorts of things. For example, here is a rule that says for *any* variable, adding 0 yields the same value:

$$\frac{\textit{Variable} \Rightarrow \textit{value}}{\textit{Variable} + 0 \Rightarrow \textit{value}}$$

Formally, the rule states that if some variable in XQuery, to which *Variable* is instantiated, evaluates to some value, to which *value* is instantiated, then we can conclude that the query '*Variable* + 0', with *Variable* substituted by the actual variable, evaluates to the same value.

We require that each unique patterns used in a particular inference rules is instantiated to the same value *for the entire rule*. This means that we can refer to "the value of *Variable*" instead of the more precise "what *Variable* is instantiated to in (this particular instantiation of) the inference rule".

Putting these together, here is an example of an inference rule that occurs later in this document:

$$\frac{\text{statEnvs} \vdash \textit{Expr}_1 : \text{xs:boolean} \quad \text{statEnvs} \vdash \textit{Expr}_2 : \textit{Type}_2 \quad \text{statEnvs} \vdash \textit{Expr}_3 : \textit{Type}_3}{\text{statEnvs} \vdash \texttt{if}\ \textit{Expr}_1\ \texttt{then}\ \textit{Expr}_2\ \texttt{else}\ \textit{Expr}_3 : (\ \textit{Type}_2 \mid \textit{Type}_3\ )}$$

This rule states that if the conditional expression of an if expression has type boolean, then the type of the entire expression is one of the two types of its then and else clauses. Note that this is represented as a choice type: '($Type_2$|$Type_3$)'.

The rule also illustrates a convention of judgment expressions: Judgments usually consist of three parts: the *assumptions*, the *problem*, and the *result*, organized as "*assumptions* |- *problem* =>> *result*" The "|-" symbol is pronounced "turnstile" and is ubiquitous in the rules whereas the actual symbol used instead of "=>>" varies with the defined judgment -- in this particular example it is the ":" symbol. The idea is that with the assumption defined one can compute the result from the problem.

> **Ed. Note:** Jonathan suggests to explain how to 'chain' inference rules. I.e., how several inference rules are applied recursively.

2.5.3 Normalization rules

Normalization is presented as logical rewrite rules, which specify those expressions that are rewritten into some other expression in the XQuery core. The static semantics is presented as type inference rules, which relate XQuery expressions to types and specify under what conditions an expression is well typed. The dynamic, or operational, semantics is presented as value inference rules, which specify the order in which an XQuery expression is evaluated and relate XQuery expressions to values.

> **Ed. Note:** I'm leaving this unchanged, but I think we can and should change the definition of normalization so that it doesn't force the use of "no-op" rules; i.e. instead of forcing the recurrent execution of more mapping rules, allow them to be pattern matched just as the rest of the

inference rules are.

Normalization applies a translation function to each expression in the XQuery syntax and returns an expression in the XQuery core syntax; The notation [*Expr*] denotes the result of translating the XQuery expression Expr into an expression in the XQuery core. All normalization rules have the following structure:

$$[\![Expr]\!] \\ == \\ coreExpr$$

The [*Expr*] above the == sign denotes an expression in the XQuery language before translation and the *coreExpr* beneath the == sign denotes an equivalent expression in the XQuery core.

For convenience, the translation of an optional expression, e.g., *Expr*?, is denoted by ([*Expr*])?, meaning that the optional expression is translated, if it exists. Similarly, the translation of a repeated expression, e.g., *Expr** is denoted by [*Expr*]*, meaning that each expression in the repetition is translated individually.

Many of the mapping rules from XQuery into the core convert expressions that have implicit arguments, types, or complex semantics into core expressions that have explicit arguments, types, and a more verbose, but explicit semantics.

2.5.4 Environments

Logical inference rules use environments to record information computed during static type inference or dynamic evaluation so that information can be used by other logical inference rules. For example, the type signature of a user-defined function in a query prolog can be recorded in an environment and used by subsequence rules. Similarly, the value assigned to a variable within a let statement can be captured in an environment and used for further evaluations.

In general an environment is a dictionary that maps a symbol (e.g., a function name or a variable name) to a value. If **env** is an environment then "env(*symbol*)" denotes the value that *symbol* is mapped to by env. Depending on the kind of environment, the "value" can be a type, a data value, etc. The notation is intentionally akin to function application as an environment can be seen as a "function" from the argument *symbol* to the value that *symbol* maps to.

In this specification we use *environment groups* that group related environments. If "envs" is an environment group with the member "mem", then that environment is denoted "envs.mem" and the value that it maps *symbol* to is denoted "envs.mem(*symbol*)".

In particular, *updating* is only defined on environment groups:

- "envs [mem(*symbol* |-> *value*)]" denotes the environment group which is just like *envs* except that the *mem* environment has been updated to map *symbol* to *value*.
- If the "value" is a type then we use the conventional variant notation "envs [mem(*symbol* : *value*)]".
- We allow the shorthand "envs [mem($symbol_1$ |-> $value_1$; ... ; $symbol_n$ |-> $value_n$)]" for the nested "(...(envs[mem($symbol_1$ |-> $value_1$)]) ... [mem($symbol_n$ |-> $value_n$)]".

Note that updating the environment overrides any previous binding that might exist for the same name. Updating the environment is used to properly capture the *scope* of variables, namespaces, etc. Also, note that

there are no operations to remove entries from environments: the need never arises because the environment group from "before" the update remains accessible concurrently with the updated version.

Finally, an example that brings several of the notations above together. Here is a rule that extends the varType member of the static environment group statEnvs with a typing of a new variable:

$$\frac{\text{statEnvs} \vdash Expr_1 : Type_1 \qquad \text{statEnvs}\,[\,\text{varType}(QName : Type_1)\,] \vdash Expr_2 : Type_2}{\text{statEnvs} \vdash \texttt{let \$}QName = Expr_1\ \texttt{return}\ Expr_2 \ : \ Type_2}$$

2.5.5 Static type inference

The result of static type inference is to associate a static type with every expression in a query, such that any evaluation of that expression is guaranteed to yield a value that belongs to that type. This section outlines how this is done.

For each type of expression (as defined by the grammar production rules of XQuery), we define *static type inference rules*. Here is a simple example of such a rule:

$$\frac{Expr_1 : Type_1 \qquad Expr_2 : Type_2}{Expr_1\ ,\ Expr_2 : Type_1, Type_2}$$

This rule is read as follows: if two expressions $Expr_1$ and $Expr_2$ have already been statically inferred to have types $Type_1$ and $Type_2$ (that's the premises above the line), then it is the case that the expression below the line "$Expr_1$, $Expr_2$" must have the type "$Type_1$, $Type_2$". Basically, it says that when two expressions are concatenated together, the result type is the concatenation of their types.

The "left half" of the expression *below* the line (the part before the ":") corresponds to some phrase in a query, for which we want to compute a type. If the query has been parsed into an internal parse tree, this usually corresponds to some node in that tree. The expression usually has *pattern*s in it ($Expr_1$ and $Expr_2$) that need to be matched against the children of the node in the parse tree. The expressions *above* the line indicate things that we need to know or compute to use this rule; in this case, we need to know the types of the two sub-expressions of the "," operator. Once we know those types (by applying static inference rules recursively to the expressions on each side), then we can compute the type of this expression. This illustrates a general feature of the type system: the type of an expression depends only on the type of its sub-expressions. The overall static type inference algorithm is a recursive, following the parse structure of the query. At each point in the recursion, an appropriate matching inference rule is sought; if at any point there is no applicable rule, then static type inference has failed: the query is not type-safe.

As stated earlier, the types computed by static inference rules are *sound*, which means that they guarantee that any value computed by an expression must belong to the static type of the expression. Another characteristic of a static type system is *completeness*. A type system is complete if it computes the "tightest possible" static type for every expression. It is very rare for type systems to be complete, and XQuery is no exception: there are expressions for which the static type system of XQuery computes a type that is not the tightest type possible.

2.5.6 Evaluation rules

Dynamic semantics associates values with query expressions. As with static type inference, we use logical

inference rules to determine the value of expressions given values of sub-expressions, and information contained in environments. XQuery's dynamic semantics is modeled on the dynamic semantics presented in [Milner].

The dynamic rules use judgments of the form: dynEnvs |- *phrase* => *value*, where *phrase* is a phrase in XQuery syntax, and *value* is a value.

The inference rules used for dynamic evaluation, like those for static type inference, follow a top-down recursive structure, computing the value of expressions from the values of their sub-expressions.

The dynamic type of data values is completely determined by the values themselves. Thus, there are no extra rules or machinery for computing dynamic types.

3 The XQuery Type System

> **Ed. Note: Status:**The XQuery type system defined in this section is based on structural typing. The definition and formal specification of the XQuery type system will be revisited when support for named typing is be added. Support for named typing is discussed in **[Issue-0104: Support for named typing]**.

The XQuery type system is used for the specification of both the dynamic and the static semantics of XQuery. It is used to describe the semantics of datatype operations and type conversion rules in XQuery. It is also the basis for the static semantics of XQuery.

The XQuery type system is based on XML Schema. XML Schema defines a notion of validation for XML documents. When doing validation, an XML Schema processor checks if the structure, and the text content of a document verifies the constraints on the structure, and the constraints on values, specified in the schema. As an example of constraints on the document structure, all `book` elements might be declared to contain an `isbn` attribute, and a `title` element, followed by a sequence of one or more `author` elements. As an example of constraints on values, the `isbn` attribute might be declared as a subset of the XML Schema `xsd:string` values, using a XML Schema pattern facet.

The XQuery type system is based on the structural constraints of XML schema, which have been formalized as [XML Schema : Formal Description]. There are two main difference between the XQuery type system and [XML Schema : Formal Description]. First the XQuery type system does not capture the notion of XML Schema *names* (also called normalized universal names in [XML Schema : Formal Description]). Second, the XQuery type system uses a specific syntax, as needed for semantics specification, notably the static type inference.

This section describes the XQuery type system, the essential operations on the XQuery type system, and the relationship between the XQuery type system and XML Schema.

3.1 XQuery Types

Types in the XQuery type system are described with the following grammar:

[75]	TypeDeclaration	::=	("define" "element" QName "{" Type? "}") | ("define" "attribute" QName "{" Type? "}") | ("define" "type" QName "{" Type? "}")

[76]	Type	::=	TypeUnion | TypeBoth | TypeSequence | TypeRepeat | ItemType | TypeRef | TypeParenthesized | TypeNone
[77]	TypeUnion	::=	Type "|" Type
[78]	TypeBoth	::=	Type "&" Type
[80]	TypeRepeat	::=	Type TypeOccurrenceIndicator
[79]	TypeSequence	::=	Type "," Type
[81]	ItemType	::=	ElementRef | AttributeRef | SimpleType
[82]	TypeRef	::=	"type" QName
[83]	TypeParenthesized	::=	"(" Type? ")"
[84]	TypeNone	::=	"none"
[55]	SimpleType	::=	QName
[85]	AttributeRef	::=	("attribute" QName) | ("attribute" NameTest "{" Type? "}")
[86]	ElementRef	::=	("element" QName) | ("element" NameTest "{" Type?)
[87]	TypeOccurrenceIndicator	::=	"*" | "+" | "?"

Ed. Note: This type system does not support text, process-instruction and document nodes. Such types have to be added (See [**Issue-0105: Types for nodes in the data model.**], and [**Issue-0068: Document Collections**]).

In addition, some constraints, not specified by the grammar, must hold:

SimpleType — *SimpleType* must be one of the *built-in simple datatypes* from [XML Schema Part 2].

The content model of a *AttributeRef* must be consistent with the allowed content of attributes. Namely, it must be a subtype of (using wilcards types as defined in section on **[3.1.1 Wildcard types]**):

AttributeRef

```
xs:AnySimpleType *
```

In the case the content of the attribute is not specified (e.g., `attribute isbn`), this indicates a reference to a globally defined attribute in the schema. Therefore the corresponding attribute must exist in the schema and be globally declared.

The content model of a *ElementRef* must be consistent with the allowed content of elements. Namely, it must be a subtype of:

ElementRef

```
( xs:AnyAttribute* ,
    xs:AnySimpleType* |
    ( xs:AnyElement | xs:string )* )
```

Note that comments and processing instructions, while they are allowed data values within an element, do not constitute part of an element's type.

In the case the content of the element is not specified (e.g., `element title`), this indicates a reference to a globally defined element in the schema. Therefore the corresponding element must exist in the schema and be globally declared.

TypeNone

The type *TypeNone* indicates a type error. This type can be generated by the type system during the course of static type analysis, but it is not a legal value in any type declaration in a query. Note that *TypeNone* is the identity for choice (I.e., (*t* | *TypeNone*) == *t*)).

Ed. Note: Jerome: The status of the constraints on attribute and element type constructors is unclear. When are they enforced and checked ? (See **[Issue-0106: Constraint on attribute and element content models]**

In [XML Schema Part 2], a *simple datatype* is either a primitive datatype, or is derived from another simple datatype by specifying a set of facets. The type syntax of XQuery does not provide any way to define datatypes with facets. Such types can be imported from XML Schemas and may be referenced as *Type* xs:QName, where xs:QName names the definition of the faceted type in the schema.

The occurrence indicators "*", "?" and "+" indicate a sequence of any, zero or one, or more than one items, respectively. This plays the same role as, but is more limited than, the minoccurs/maxoccurs facets of [XML Schema Part 1], and follows the notations of DTDs.

A *WildCard* denotes a set of names. A *WildCard* is either

- a *QName* denoting the name with the given namespace prefix and local name;

- * denoting any name in any namespace;
- *NCName*:* denoting any local name in namespace *NCName*;
- or *:*NCName* denoting the local name *NCName* in any namespace;

The `element`, and `attribute`, type constructors may use a wildcard to specify sets of one or more matching elements, or attributes. For example,

```
element bib:* { xs:integer* }
```

indicates any element in the `bib` namespace whose content consists of zero or more integers.

Empty content is indicated with the explicit empty sequence, as in

```
element bib:* { ( ) }
```

If the content model is omitted, then the element, (resp. attribute or type) name must be a proper *QName*, and refers to the global element (resp. attribute or type) with that name, which must be defined (by importation from a schema or through a type definition expression).

The type system includes three operators that can combine types: ",", "|" and "&".

The "," operator builds the "sequence" of two types. For example,

```
xs:integer , xs:double
```

means a sequence of an integer followed by a double. This corresponds to the xs:sequence construct of XML Schema for elements; XML Schema does not recognize similar properties for simple types.

The "|" operator builds a "choice" between two types. For example,

```
xs:integer | (element bib:author)*
```

means either an integer or a sequence of `bib:author` elements. The "|" operator corresponds to the xs:choice construct of XML Schema for elements and the xs:union construct for simple types.

The "&" operator builds the "interleaved product" of two types. The type t_1 & t_2 matches any sequence that is an interleaving of a sequence that matches t_1 and a sequence that matches t_2. For example,

```
(element a , element b) & xs:string   =
                element a, element b, xs:string
              | element a, xs:string, element b
              | xs:string, element a, element b
```

In each case, the order of `element a` and `element b` is unchanged, but the xs:string can be interleaved in any position.

The interleaved product is a generalization of the xs:all construct in XML Schema for elements, and has no corresponding representation in XML Schema for simple types. The xs:all construct in XML Schema may only consist of global or local element declarations with lower bound 0 or 1, and upper bound 1.

3.1.1 Wildcard types

XQuery defines the following built-in datatypes:

```
define type xs:AnySimpleType { xs:integer | xs:float | xs:string ... }
define type xs:AnyAttribute  { attribute * { type xs:AnySimpleType * }}
define type xs:AnyElement    { element *  { type xs:AnyType } }
define type xs:AnyNode       { type xs:AnyAttribute | type xs:AnyElement }
define type xs:AnyItem       { type xs:AnySimpleType | type xs:AnyNode }
define type xs:AnyType       { type xs:AnyItem* }
```

The concepts of xs:AnySimpleType and xs:AnyType have the same interpretation as anySimpleType and anyType of XML Schema, respectively, though this version of xs:AnyType contains types (such as comments and processing instructions) that do not occur in XML Schema.

3.2 Instances, and Domain of a Type

> **Ed. Note: Status:**This section is still incomplete and based on structural typing. The notion of domain of a type will be revised when support for named typing is added.

The semantics of a type is given by the notion of domain, I.e., the set of all values which are instance of that type. A tree in the [XQuery 1.0 and XPath 2.0 Data Model] is an instance of a type in the XQuery type system, if and only if it verifies the structural constraints described by the type. For instance the query:

```
<bib:pubdate> { 1954, 1966, 1974, 1986 } </bib:pubdate>
```

creates an instance of the following type:

```
element bib:* { attribute country { xs:string }?, xs:integer* }
```

because the element name "bib:pubdate" is an instance of the wildcard nametest "bib:*", the attribute "country" is optional, and the content of that element is indeed a sequence of integers.

The notion of instance of a type in XQuery is structural. Notably, it does not take XML Schema names into account. For instance, the two following XQuery types:

```
define element entry { type Book }
define type Book {
  attribute isbn { xs:string },
  element title { xs:string },
  element author { xs:string }*,
}
```

and

```
define element entry { type Article }
define type Article {
  attribute isbn { xs:string }?
  (element title { xs:string } | element ref { xs:string }),
  element author { xs:string }+,
}
```

might be declared in two different schemas, one for books and one for research articles. Because the notion of instance in the XQuery type system is structural, the following document:

```
<entry> isbn="0-618-15398-5">
  <title>The Lord of the Rings</title>
  <author>J.R.R. Tolkien</author>
</entry>
```

would always be an instance of both types "Book" and "Article". In XML Schema, depending against which schema the document is validated, only one of the two above types would hold.

In this section, we define formally the domain of a XQuery type. We first define the domain of a nametest, then define the domain of a complete type.

3.2.1 Domain of a NameTest

The semantics of a given NameTest is defined by the set of names that matches a that NameTest.

Notation

We denote access to the domain of a NameTest with the following auxiliary judgment. The notation:

$$\text{statEnvs} \vdash \mathit{QName}_1 \ \texttt{in}_{\text{wild}} \ \mathit{NameTest}_2$$

indicates that the QName *QName* is in the domain defined by the NameTest $\mathit{NameTest}_2$.

We specify the relationship $\mathit{QName}_1 \ \texttt{in}_{\text{wild}} \ \mathit{NameTest}_2$ by decomposing both sides in to *QName*s or *WildCard*s, and then defining the relationship between each component of the name.

First, is always in the domain of *QName* is always a singleton compose of itself. Note that the following inference rule takes namespace resolution into account.

$$\frac{\begin{array}{rr}\text{statEnvs} \vdash \texttt{expand}(\mathit{QName}_1) \Rightarrow \mathit{qname}(\mathit{URI}_1,\mathit{ncname}_1) & \mathit{URI}_1 = \mathit{URI}_2 \\ \text{statEnvs} \vdash \texttt{expand}(\mathit{QName}_2) \Rightarrow \mathit{qname}(\mathit{URI}_2,\mathit{ncname}_2) & \mathit{ncname}_1 = \mathit{ncname}_2\end{array}}{\text{statEnvs} \vdash \mathit{QName}_1 \ \texttt{in}_{\text{wild}} \ \mathit{QName}_2}$$

$$\frac{\begin{array}{c}\text{statEnvs} \vdash \texttt{expand}(\mathit{QName}_1) \Rightarrow \mathit{qname}(\mathit{ncname}_1) \\ \text{statEnvs} \vdash \texttt{expand}(\mathit{QName}_2) \Rightarrow \mathit{qname}(\mathit{ncname}_2) \qquad \mathit{ncname}_1 = \mathit{ncname}_2\end{array}}{\text{statEnvs} \vdash \mathit{QName}_1 \ \texttt{in}_{\text{wild}} \ \mathit{QName}_2}$$

Now the rules for $\mathit{QName} \ \texttt{in}_{\text{wild}} \ \mathit{WildCard}$:

$$\frac{\begin{array}{c}\text{statEnvs} \vdash \texttt{expand}(\mathit{QName}_1) \Rightarrow \mathit{qname}(\mathit{URI}_1,\mathit{ncname}_1) \qquad \mathit{URI}_1 = \mathit{URI}_2 \\ \text{statEnvs} \vdash \texttt{expandNCName}(\mathit{NCName}_2) \Rightarrow \mathit{URI}_2\end{array}}{\text{statEnvs} \vdash \mathit{QName}_1 \ \texttt{in}_{\text{wild}} \ \mathit{NCName}_2\texttt{:*}}$$

$$\frac{\text{statEnvs} \vdash \texttt{expand}(\mathit{QName}_1) \Rightarrow \mathit{qname}(\mathit{URI}_1,\mathit{ncname}_1) \qquad \mathit{ncname}_1 = \mathit{NCName}_2}{\text{statEnvs} \vdash \mathit{QName}_1 \ \texttt{in}_{\text{wild}} \ \texttt{*:}\mathit{NCName}_2}$$

$$\frac{}{\text{statEnvs} \vdash \mathit{QName}_1 \ \texttt{in}_{\text{wild}} \ \texttt{*}}$$

3.2.2 Semantics of the Domain of a Type

Notation

We denote the domain of a type using the following auxiliary judgment. The notation:

statEnvs |- *value* `in` *Type*

indicates that the value is the domain of type *Type*.

The XQuery type system is based on tree grammars. The semantics for the domain of a type is based on the set of a trees accepted by the corresponding tree grammar.

We do now give a full specification of the notion of language recognized by a tree grammar at this point, but refer the reader to the literature on tree grammars, for instance [Languages], or [TATA]

> **Ed. Note:** Future versions of that document will include a more complete description of the notion of domain of a type.
>
> **Ed. Note:** The use of pure tree grammars for defining the domain yields a structural definition of the domain of a type. This definition will have to be revised as soon as the named typing proposal is added.

3.3 Subtyping

This section defines the semantics of subtyping in XQuery. Subtyping is an important relationship between type, which is used notably, when doing function applications to check whether the input parameters of the function are valid.

We first define subtyping between nametests, then define the notion of subtyping between arbitrary types.

3.3.1 NameTest Subtyping

Notation

We denote NameTest subtyping with the following auxiliary judgment. The notation:

$$\text{statEnvs} \vdash \mathit{NameTest}_1 <:_{\text{wild}} \mathit{NameTest}_2$$

indicates that NameTest $\mathit{NameTest}_1$ is a subtype of the NameTest $\mathit{NameTest}_2$.

We evaluate $\mathit{NameTest}_1 <:_{\text{wild}} \mathit{NameTest}_2$ by decomposing both sides in to *QName*s or *WildCard*s, and then defining a subtyping relation between each component of the name.

First $\mathit{QName} <:_{\text{wild}} \mathit{QName}$ is only true if they represent the same fully qualified name:

$$\frac{\begin{array}{rl}\text{statEnvs} \vdash \texttt{expand}(\mathit{QName}_1) \Rightarrow \mathit{qname}(\mathit{URI}_1,\mathit{ncname}_1) & \mathit{URI}_1 = \mathit{URI}_2 \\ \text{statEnvs} \vdash \texttt{expand}(\mathit{QName}_2) \Rightarrow \mathit{qname}(\mathit{URI}_2,\mathit{ncname}_2) & \mathit{ncname}_1 = \mathit{ncname}_2\end{array}}{\text{statEnvs} \vdash \mathit{QName}_1 <:_{\text{wild}} \mathit{QName}_2}$$

$$\frac{\text{statEnvs} \vdash \texttt{expand}(QName_1) \Rightarrow qname(ncname_1) \quad \text{statEnvs} \vdash \texttt{expand}(QName_2) \Rightarrow qname(ncname_2) \quad ncname_1 = ncname_2}{\text{statEnvs} \vdash QName_1 <:_{\text{wild}} QName_2}$$

Now the rules for $QName <:_{\text{wild}} WildCard$:

$$\frac{\text{statEnvs} \vdash \texttt{expand}(QName_1) \Rightarrow qname(URI_1, ncname_1) \quad URI_1 = URI_2 \quad \text{statEnvs} \vdash \texttt{expandNCName}(NCName_2) \Rightarrow URI_2}{\text{statEnvs} \vdash QName_1 <:_{\text{wild}} NCName_2 : *}$$

$$\frac{\text{statEnvs} \vdash \texttt{expand}(QName_1) \Rightarrow qname(URI_1, ncname_1) \quad ncname_1 = NCName_2}{\text{statEnvs} \vdash QName_1 <:_{\text{wild}} * : NCName_2}$$

$$\frac{}{\text{statEnvs} \vdash QName_1 <:_{\text{wild}} *}$$

And finally rules for $WildCard_1 <:_{\text{wild}} WildCard_2$. Note that the second rule also prevents $NCName : *$ from being a subtype of anything unless it is a valid known prefix.

$$\frac{}{\text{statEnvs} \vdash * <:_{\text{wild}} *}$$

$$\frac{\texttt{expandNCName}(NCName) \Rightarrow URI}{\text{statEnvs} \vdash NCName : * <:_{\text{wild}} *}$$

$$\frac{}{\text{statEnvs} \vdash * : NCName <:_{\text{wild}} *}$$

> **Ed. Note:** DD: (1) I've taken liberties comparing, e.g. *NCName* = *ncname*, (2) this doesn't take into account any resolution of URIs.

3.3.2 Semantics of Subtyping

Notation

We denote subtyping with the following auxiliary judgment. The notation:

$$\text{statEnvs} \vdash Type_1 <: Type_2$$

indicates that type $Type_1$ is a subtype of $Type_2$.

The definition of subtyping is based on the notion of domain, as defined in **[3.2 Instances, and Domain of a Type]**

By definition, $Type_1 <: Type_2$ if and only if all the instances in the domain of $Type_1$ are also in the domain of

$Type_2$. That is, for every value *value* such that:

statEnvs |- *value* `in` $Type_1$

it is also the case that:

statEnvs |- *value* `in` $Type_2$

It is easy to see that the subtype relation `<:` is a partial order, i.e. it is **reflexive**: statEnvs |- *Type* `<:` *Type*

and it is **transitive**: if, statEnvs |- $Type_1$ `<:` $Type_2$ and, statEnvs |- $Type_2$ `<:` $Type_3$ then, statEnvs |- $Type_1$ `<:` $Type_3$

Note

The above definition although complete and precise, does not give a simple means to *compute* subtyping.

The XQuery type system is based on tree grammars. Computing subtyping between two types can be done by computing inclusion between their corresponding tree grammar.

We do now give a full algorithm to compute inclusion between tree grammar at this point, but refer the reader to the literature on tree grammars, for instance [Languages], or [TATA]

> **Ed. Note:** Future versions of that document will include a more complete description of algorithms to compute subtyping.
>
> **Ed. Note:** The use of pure tree grammars for defining the domain yields a structural definition of the notion of subtyping. This definition will have to be revised as soon as the named typing proposal is added.

3.3.3 Type equivalence

In a few cases, we use the fact that two types are *equivalent*, I.e., that they define exactly the same domain over data model instances.

By definition, we write $Type_1$ = $Type_2$ if and only if, $Type_1$`<:`$Type_2$ and $Type_2$`<:` $Type_1$.

3.4 Prime types

This section defines the notion of *Prime Type*, and operations on prime types. Prime types play an important role in defining the static semantics of XQuery.

3.4.1 Prime types and sequences of items with similar types

Some expressions operate on sequences of similar items. These include FLWR expressions, sorting, and duplicate elimination. For example, consider the following small document:

```
<paper>
    <author>Buneman</author>
    <author>Davidson</author>
    <author>Fernandez</author>
    <author>Suciu</author>
    <title>Adding structure to unstructured data</title>
    <editor>Afrati</editor>
```

```
      <editor>Kolaitis</editor>
   </paper>
:  element paper {
      element author {xsd:string}+,
      element title {xsd:string},
      element editor {xsd:string}*
   }
```

The following query extracts a sequence of authors followed by editors and sorts them by content. This results in a sequence with authors and editors arbitrarily interleaved. The result type reflects this by repeating a choice of author and editor.

```
(/paper/author , /paper/editor) sortby .
 => <editor>Afrati</editor>
    <author>Buneman</author>
    <author>Davidson</author>
    <author>Fernandez</author>
    <editor>Kolaitis</editor>
    <author>Suciu</author>
 :  (element author {xsd:string} | element editor {xsd:string})+
```

Such operation always operate on sequences over choices of items, that we call a *Prime type*.

Prime types can be described by the following grammar production.

[88] PrimeType ::= ItemType
| (PrimeType "|" PrimeType)

This section explains the details about computing over prime types and sequences of prime types. For instance, we explain why the above type ends in a plus rather than a star.

3.4.2 Computing Prime Types

We use two auxiliary functions on types. Note that both functions are only used by XQuery's type inference rules; they are not part of the XQuery syntax.

The type function prime(*Type*) extracts all item types from the type expression *Type* and combines them by a choice.

prime(*ItemType*)	=	*ItemType*
prime(())	=	none
prime(none)	=	none
prime($Type_1$, $Type_2$)	=	prime($Type_1$) | prime($Type_2$)
prime($Type_1$ & $Type_2$)	=	prime($Type_1$) | prime($Type_2$)
prime($Type_1$ | $Type_2$)	=	prime($Type_1$) | prime($Type_2$)
prime(*Type*?)	=	prime(*Type*)
prime(*Type**)	=	prime(*Type*)
prime(*Type*+)	=	prime(*Type*)

The type function quantifier(*Type*) approximates the possible number of items in *Type* with the quantifiers

supported by XQuery type expressions (`?, +, *`). To quantify interim results, we use the auxiliary quantifiers `1` for exactly one occurrence, `0` for exactly zero occurrences, i.e., the quantifier of the empty list, and `-` for infinitely many occurrences, i.e., the quantifier of the empty choice `none`, which is used to type errors.

quantifier(*ItemType*)	=	1
quantifier(())	=	0
quantifier(none)	=	0
quantifier($Type_1$, $Type_2$)	=	quantifier($Type_1$) , quantifier($Type_2$)
quantifier($Type_1$ & $Type_2$)	=	quantifier($Type_1$) , quantifier($Type_2$)
quantifier($Type_1$ | $Type_2$)	=	quantifier($Type_1$) | quantifier($Type_2$)
quantifier(*Type*?)	=	quantifier(*Type*) · ?
quantifier(*Type**)	=	quantifier(*Type*) · *
quantifier(*Type*+)	=	quantifier(*Type*) · +

The tables below specify the necessary arithmetics to compute the sum (q_1 , q_2), the choice (q_1 | q_2), and the product ($q_1 \cdot q_2$) of two quantifiers q_1, q_2.

,	0	1	?	+	*
0	0	1	?	+	*
1	1	+	+	+	+
?	?	+	*	+	*
+	+	+	+	+	+
*	*	+	*	+	*

|	0	1	?	+	*
0	0	?	?	*	*
1	?	1	?	+	*
?	?	?	?	*	*
+	*	+	*	+	*
*	*	*	*	*	*

·	0	1	?	+	*
0	0	0	0	0	0
1	0	1	?	+	*
?	0	?	?	*	*
+	0	+	*	+	*
*	0	*	*	*	*

The auxiliary quantifiers for interim results are eliminated with the following type simplification rules.

Type · 0	=	()
Type · 1	=	*Type*
Type · ?	=	*Type*?
Type · +	=	*Type*+
Type · *	=	*Type**
Type · -	=	none

prime(*Type*) · quantifier(*Type*) enjoys two desirable properties:

(1) for all types $Type_1$, $Type_2$: $Type_1$ <: $Type_2$ implies prime($Type_1$) · quantifier($Type_1$) <: prime($Type_2$) · quantifier($Type_2$).
For example, for the two syntactically different but semantically equivalent types $Type_1$ `= (xsd:string & xsd:integer)*` and $Type_2$ `=((xsd:string,xsd:integer) | (xsd:integer,xsd:string))*`, prime($Type_1$) · quantifier($Type_1$) = prime($Type_2$) · quantifier($Type_2$) = `(xsd:string | xsd:integer)*`.

(2) for all types *Type*: *Type* <: prime(*Type*) · quantifier(*Type*).
Note that for some type expressions prime(*Type*) · quantifier(*Type*) may be regarded as too general. For example, the result-type of sorting a sequence of type `xsd:integer , xsd:string` is `(xsd:integer | xsd:string)+`. A more specific result-type would be `xsd:integer & xsd:string`, which preserves the individual quantifiers for `xsd:integer` and `xsd:string`. However,

computing such result-types for arbitrary type expressions loses type precision for choice types (e.g. `(xsd:integer | xsd:string)+` becomes `(xsd:integer* & xsd:string*)`), and it requires more complicated typing rules.

> **Ed. Note:** (Peter): I'm not sure whether we need to elaborate on this here, but at least Denise may wonder why we ditched disorder(). In addition, I'm aware that (2) is a corollary of Phil's famous factoring theorem, I'm not sure whether this also holds for (1), but haven't spend much thought on it. BTW. Impotence holds of course as well.

Using these two auxiliary functions, the result type of a *SortExpr* can be computed as follows. If *Expr* has type *Type* then *Expr* sortby *SortSpecList* has type prime(*Type*) · quantifier(*Type*). For the formal type inference rule see **[4.10 Sorting Expressions]**. Similar rules are applied to type the other operations that destroy sequence order - union, intersect, except, distinct-node, distinct-value, and unorder.

> **Ed. Note:** (Peter): I deliberately do not introduce the formal inference rule here, because (a) the machinery for type inference probably hasn't been explained yet at this place in the document, and (b) the current static semantics of *SortExpr* is incomplete anyway. Maybe the example should use unorder() instead of sortby.

3.4.3 Common Prime Types

> **Ed. Note: Status:** This section contains new material necessary for the simplified semantics of typeswitch.

When performing type checking for certain XQuery expressions, we also need to compute the common prime types between two types, and the common occurrence indicator between two types.

The type function common-primes($Type_1$,$Type_2$) extracts all item types that are common from the prime types and and combines them into a new prime type.

Notation

In order to define the "common-primes" function, we use the following auxiliary judgment.

$$\text{statEnvs} \vdash \text{common-primes}(ItemType_1, ItemType_2) = Type$$

specifies that under the current static environment, the common prime of $ItemType_1$, $ItemType_2$ is type *Type*.

We define the common primes by looking at each combination of items in each prime type.

$$\text{common-primes}((ItemType_1 \mid \cdots \mid ItemType_n), (ItemType_1' \mid \cdots \mid ItemType_r'))$$
$$=$$
$$\text{common-primes}(ItemType_1, ItemType_1') \mid \cdots \mid \text{common-primes}(ItemType_n, ItemType_r')$$

We then we define whether each item is kept or discarded using the auxiliary judgment. In the case the item is discarded, the result of common-primes is none. Remember that none is the unit for choice.

In the case of a simple type, or globally defined attribute or element, one keeps the corresponding item if both items in the functions are the same.

$$\frac{\text{statEnvs} \vdash SimpleType_1 = SimpleType_2}{\text{statEnvs} \vdash \text{common-primes}(SimpleType_1, SimpleType_2) = SimpleType_1}$$

$$\frac{\text{statEnvs} \text{ |- element } QName_1 = \text{element } QName_2}{\text{statEnvs} \text{ |- common-primes(element } QName_1, \text{ element } QName_2) = \text{element } QName_1}$$

$$\frac{\text{statEnvs} \text{ |- attribute } QName_1 = \text{attribute } QName_2}{\text{statEnvs} \text{ |- common-primes(attribute } QName_1, \text{ attribute } QName_2) = \text{attribute } QName_1}$$

When matching an element (resp. attribute), if the left-hand side item type is a subtype of the right-hand side item type, then the function returns the left-hand side item type. This rule is particularly useful, when the right hand side is a wildcard type, as might be often the case in a typeswitch expression.

$$\frac{\text{statEnvs} \text{ |- element } NameTest_1 \{ Type_1 \} <: \text{element } NameTest_2 \{ Type_2 \}}{\text{statEnvs} \text{ |- common-primes(element } NameTest_1 \{ Type_1 \}, \text{ element } NameTest_2 \{ Type_2 \}) = \text{element } NameTest_1 \{ Type_1 \}}$$

$$\frac{\text{statEnvs} \text{ |- element } NameTest_1 \{ Type_1 \} <: \text{element } QName_2}{\text{statEnvs} \text{ |- common-primes(element } NameTest_1 \{ Type_1 \}, \text{ element } QName_2) = \text{element } NameTest_1 \{ Type_1 \}}$$

$$\frac{\text{statEnvs} \text{ |- element } QName_1 <: \text{element } NameTest_2 \{ Type_2 \}}{\text{statEnvs} \text{ |- common-primes(element } QName_1, \text{ element } NameTest_2 \{ Type_2 \}) = \text{element } QName_1}$$

$$\frac{\text{statEnvs} \text{ |- attribute } NameTest_1 \{ Type_1 \} <: \text{attribute } NameTest_2 \{ Type_2 \}}{\text{statEnvs} \text{ |- common-primes(attribute } NameTest_1 \{ Type_1 \}, \text{ attribute } NameTest_2 \{ Type_2 \}) => \text{attribute } NameTest_1 \{ Type_1 \}}$$

$$\frac{\text{statEnvs} \text{ |- attribute } NameTest_1 \{ Type_1 \} <: \text{attribute } QName_2}{\text{statEnvs} \text{ |- common-primes(attribute } NameTest_1 \{ Type_1 \}, \text{ attribute } QName_2) => \text{attribute } NameTest_1 \{ Type_1 \}}$$

$$\frac{\text{statEnvs} \text{ |- attribute } QName_1 <: \text{attribute } NameTest_2 \{ Type_2 \}}{\text{statEnvs} \text{ |- common-primes(attribute } QName_1, \text{ attribute } NameTest_2 \{ Type_2 \}) => \text{attribute } QName_1}$$

Otherwise, if the left-hand side item and the right-hand side item have compatible names, then the function returns the right-hand side item type.

$$\frac{\begin{array}{c}\text{statEnvs} \text{ |- not (element } NameTest_1 \{ Type_1 \} <: \text{element } NameTest_2 \{ Type_2 \}) \\ \text{statEnvs} \text{ |- } (NameTest_1 <:_{\text{wild}} NameTest_2) \text{ or } (NameTest_2 <:_{\text{wild}} NameTest_1)\end{array}}{\text{statEnvs} \text{ |- common-primes(element } NameTest_1 \{ Type_1 \}, \text{ element } NameTest_2 \{ Type_2 \}) => \text{element } NameTest_2 \{ Type_2 \}}$$

$$\frac{\text{statEnvs} \text{ |- not(element } NameTest_1 \text{ \{ } Type_1 \text{ \} <: element } QName_2 \text{)} \quad \text{statEnvs} \text{ |- } (NameTest_1 <:_{\text{wild}} QName_2) \text{ or } (QName_2 <:_{\text{wild}} NameTest_1)}{\text{statEnvs} \text{ |- common-primes(element } NameTest_1 \text{ \{ } Type_1 \text{ \}, element } QName_2) \Rightarrow \text{element } QName_2}$$

$$\frac{\text{statEnvs} \text{ |- not(element } QName_1 \text{ <: element } NameTest_2 \text{ \{ } Type_2 \text{ \})} \quad \text{statEnvs} \text{ |- } (QName_1 <:_{\text{wild}} NameTest_2) \text{ or } (NameTest_2 <:_{\text{wild}} QName_1)}{\text{statEnvs} \text{ |- common-primes(element } QName_1, \text{ element } NameTest_2 \text{ \{ } Type_2 \text{ \}) => element } NameTest_2 \text{ \{ } Type_2 \text{ \}}$$

$$\frac{\text{statEnvs} \text{ |- not(attribute } NameTest_1 \text{ \{ } Type_1 \text{ \} <: attribute } NameTest_2 \text{ \{ } Type_2 \text{ \})} \quad \text{statEnvs} \text{ |- } (NameTest_1 <:_{\text{wild}} NameTest_2) \text{ or } (NameTest_2 <:_{\text{wild}} NameTest_1)}{\text{statEnvs} \text{ |- common-primes(attribute } NameTest_1 \text{ \{ } Type_1 \text{ \}, attribute } NameTest_2 \text{ \{ } Type_2 \text{ \}) => attribute } QName_2 \text{ \{ } Type_2 \text{ \}}$$

$$\frac{\text{statEnvs} \text{ |- not(attribute } NameTest_1 \text{ \{ } Type_1 \text{ \} <: attribute } QName_2 \text{)} \quad \text{statEnvs} \text{ |- } (NameTest_1 <:_{\text{wild}} QName_2) \text{ or } (QName_2 <:_{\text{wild}} NameTest_1)}{\text{statEnvs} \text{ |- common-primes(attribute } NameTest_1 \text{ \{ } Type_1 \text{ \}, attribute } QName_2) \Rightarrow \text{attribute } QName_2}$$

$$\frac{\text{statEnvs} \text{ |- not(attribute } QName_1 \text{ <: attribute } NameTest_2 \text{ \{ } Type_2 \text{ \})} \quad \text{statEnvs} \text{ |- } (QName_1 <:_{\text{wild}} NameTest_2) \text{ or } (NameTest_2 <:_{\text{wild}} QName_1)}{\text{statEnvs} \text{ |- common-primes(attribute } QName_1, \text{ element } NameTest_2 \text{ \{ } Type_2 \text{ \}) => attribute } NameTest_2 \text{ \{ } Type_2 \text{ \};}$$

Finally, in all the other cases, the common type is none. I.e., otherwise:

$$\text{statEnvs} \text{ |- common-primes}(ItemType_1, ItemType_2) \Rightarrow \text{none}$$

3.4.3.1 Common Occurrence Indicator

The following table is computing the common occurrence indicator and is used in the typing of the typeswitch expression.

common-occur	0	1	?	+	*
0	0	0	0	0	0
1	0	1	1	1	1
?	0	1	?	1	?
+	0	1	1	+	+
*	0	1	?	+	*

3.5 Importing types from XML Schema

> **Ed. Note: Status:** This section is still incomplete, needs detailed review, and will be fully revised to support named typing. This section is enclose here merely to illustrate what mapping from XML Schema into the XQuery type system will look like in future versions of the document.

The XQuery type system support a static subset of XML Schema. At compile time, the XQuery environment imports XML Schema declarations and load them as declarations in the XQuery type system for further static reasoning. The semantics of that loading process is defined by a simple mapping from XML Schema to the XQuery type system.

The mapping applies a translation function to each XML Schema expression in the XML Schema syntax and returns an expression in the XQuery type system syntax. The notation [*SchemaExpr*] denotes the result of translating the XML Schema expression Expr into an expression in the XQuery type system. All mapping rules have the following structure:

$$\frac{[\![\ \mathit{SchemaExpr}\ ]\!]}{\mathit{Type}}$$

The [*SchemaExpr*] above the horizontal rule denotes an expression in XML Schema before translation and the *coreExpr* beneath the horizontal rule denotes an equivalent expression in the XQuery core.

3.5.1 Schema

Schemas are imported by the 'schema' declaration in the preamble of a query. To import a schema, the document referred to by the given URI is opened and the schema declarations contained in the document are translated into the corresponding in-line type definitions.

$$\frac{[\![\ \text{schema NCName \{ URI \}}\ ]\!]}{[\![\ \text{open-schema-document(URI)}\ ]\!]_{\text{sort\_decl(NCName)}}}$$

To make the translation rules a bit easier to read, we use grammar rules to distinguish between the various parts of a <schema> element:

[89]	Pragma	::=	("include" | "import" | "redefine" | "annotation")*
[90]	Content	::=	(("simpleType" | "complexType" | "element" | "attribute" | "attributeGroup""attributeGroup" | "group" | "notation") "annotation"*)*

(Note: attributes that are ignored are emphasized)

$$[$$

<schema
attributeFormDefault = (qualified | unqualified) : unqualified
elementFormDefault = (qualified | unqualified) : unqualified
finalDefault = (#all | List of (extension | restriction)) : ''
id = ID
targetNamespace = anyURI
version = token
xml:lang = language
{any attributes with non-schema namespace . . .}>
Pragma Content
</schema>

$$]_{\text{sort\_decl(NS)}} = [\text{Pragma}]_{\text{sort\_decl(NS)}}\ [\text{Content}]_{\text{sort\_decl(NS)}}$$

$$[$$

<include
id = ID
schemaLocation = anyURI
{any attributes with non-schema namespace . . .}>
Content: (annotation?)
</include>

$$]$$

$$= [\text{schema targetNS \{ anyURI \}}]_{\text{sort\_decl(targetNS)}}$$

An imported schema is associated with the namespace prefix that it defines:

$$[$$

```
<import
    id = ID
    namespace = NCName
    schemaLocation = anyURI
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?)
</import>
```

$$]$$

$$==$$

$$[\text{schema NCName \{ anyURI \}}]_{\text{sort\_decl(namespace)}}$$

Redefine is unsupported.

$$[$$

```
<redefine
    id = ID
    schemaLocation = anyURI
    {any attributes with non-schema namespace . . .}>
    Content: (annotation | (simpleType | complexType | group | attributeGroup))*
</redefine>
```

$$]_{\text{sort\_decl(targetNS)}}$$

$$==$$

Annotations are unsupported.

$$[\text{annotation}]_{\text{sort\_decl(targetNS)}}$$

$$==$$

3.5.2 QNames

$$[\text{NCName}]_{\text{name(targetNS)}}$$

$$==$$

$$\text{targetNS:NCName}$$

$$[\text{NS:NCName}]_{\text{name(targetNS)}}$$

==

NS:NCName

3.5.3 Complex Type Definitions

The following productions describe the structure of a complex type definition in XML Schema.

[91]	ComplexTypeContent	::=	"annotation"? ("simpleContent" | "complexContent" | "group" | "all" | "choice" | "sequence")
[92]	Attributes	::=	("attribute" | "attributeGroup""attributeGroup")* "anyAttribute"?

```
<complexType
   abstract = boolean : false
   block = (#all | List of (extension | restriction))
   final = (#all | List of (extension | restriction))
   id = ID
   mixed = boolean : false
   name = NCName
   {any attributes with non-schema namespace . . .}>
   ComplexTypeContent
   Attributes
</complexType>
```

We distinguish between global complex types (which are mapped to sort declarations) and local complex types (which are mapped to type definitions).

3.5.3.1 Global complex type

$$\left[\begin{array}{l}\text{<complexType name=NCName>}\\ \quad\text{ComplexTypeContent}\\ \quad\text{Attributes}\\ \text{</complexType>}\end{array}\right]_{\text{sort\_decl(prefix)}}$$

==

$$\text{define type } [\text{NCName}]_{\text{name(prefix)}} \{ [\text{Attributes}]_{\text{group(prefix)}}, [\text{ComplexTypeContent}]_{\text{group(prefix)}} \}$$

In case of mixed complex type, the mapping introduces TEXT in the corresponding type system representation.

$$\left[\begin{array}{l}\text{<complexType name=NCName mixed="true">}\\ \quad \text{ComplexTypeContent}\\ \quad \text{Attributes}\\ \text{</complexType>}\end{array}\right]_{\text{sort\_decl(prefix)}}$$

$$==$$

$$\text{define type } \left[\text{NCName}\right]_{\text{name(prefix)}} \{ \left[\text{Attributes}\right]_{\text{group(prefix)}}, \text{TEXT*} \& \left[\text{ComplexTypeContent}\right]_{\text{group(prefix)}} \}$$

3.5.3.2 Anonymous local complex type

$$\left[\begin{array}{l}\text{<complexType>}\\ \quad \text{ComplexTypeContent}\\ \quad \text{Attributes}\\ \text{</complexType>}\end{array}\right]_{\text{sort\_decl(prefix)}}$$

$$==$$

$$\left[\text{Attributes}\right]_{\text{group(prefix)}}, \left[\text{ComplexTypeContent}\right]_{\text{group(prefix)}}$$

In case of mixed complex type, the mapping introduces TEXT in the corresponding type system representation.

$$\left[\begin{array}{l}\text{<complexType mixed="true">}\\ \quad \text{ComplexTypeContent}\\ \quad \text{Attributes}\\ \text{</complexType>}\end{array}\right]_{\text{sort\_decl(prefix)}}$$

$$==$$

$$\left[\text{Attributes}\right]_{\text{group(prefix)}}, \text{TEXT*} \;\&\left[\text{ComplexTypeContent}\right]_{\text{group(prefix)}}$$

3.5.4 Groups

3.5.4.1 Global named group declarations

$$\left[\right.$$

```
<group
   name = NCName>
   Content: (annotation?, (all | choice | sequence))
</group>
```

$$\left]_{\text{group(prefix)}} = =\right.$$

$$\text{define type } \left[\text{NCName}\right]_{\text{name(prefix)}} \{ \left[\text{Content}\right]_{\text{group(prefix)}} \}$$

3.5.4.2 Local groups

Anonymous local groups

$$\left[\right.$$

```
<group>
   Content: (annotation?, (all | choice | sequence))
</group>
```

$$\left]_{\text{sort\_decl(prefix)}} = =\right.$$

$$\left[\text{Content}\right]_{\text{group(prefix)}}$$

All groups

$$[$$

```
<all
   id = ID
   maxOccurs = 1 : 1
   minOccurs = (0 | 1) : 1
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, element1 ... elementn)
</all>
```

$$]_{group(prefix)} == [element_1]_{group(prefix)} \& ... \& [element_n]_{group(prefix)}$$

Choice groups

$$[$$

```
<choice
   id = ID
   Occurs
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (element | group | choice | sequence | any)*)
</choice>
```

$$]_{group(prefix)} == ([Content_1]_{group(prefix)} | ... | [Content_n]_{group(prefix)}) [Occurs]_{bound}$$

Sequence groups

$$[$$

```
<sequence
   id = ID
   Occurs
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (element | group | choice | sequence | any)*)
</sequence>
```

$$\left]_{group(prefix)}\right.$$
==
$$(\left[Content_1\right]_{group(prefix)}, \ldots, \left[Content_n\right]_{group(prefix)})\ \left[Occurs\right]_{bound}$$

3.5.5 minOccurs and maxOccurs

The following structure describes minOccurs and maxOccurs attributes in XML Schema.

[93] Occurs ::= ("maxOccurs" ("nonNegativeInteger" | "unbounded"))
| ("minOccurs" "nonNegativeInteger")

minOccurs and maxOccurs are mapped to the type system in the following way.

$$\left[minOccurs=0\right]_{bound}$$
==
?

$$\left[maxOccurs="unbounded"\right]_{bound}$$
==
+

$$\left[minOccurs=0\ maxOccurs="unbounded"\right]_{bound}$$
==
*

$$\left[minOccurs=1\ maxOccurs=1\right]_{bound}$$
==

Right now, the proposed type system grammar does not capture other occurrences cases. Therefore, these are converted into a collection.

$$\left[minOccurs=m\ maxOccurs=n\right]_{bound}$$
==
*

3.5.6 Elements

The following structure describes element declarations in XML Schema.

<element
abstract = boolean : false
block = (#all | List of (extension | restriction | substitution))
default = string
final = (#all | List of (extension | restriction))
fixed = string

form = (qualified | unqualified)
id = ID
Occurs
name = NCName
nillable = boolean : false
ref = QName
substitutionGroup = QName
type = QName
{any attributes with non-schema namespace . . .}>
Content: (annotation?, ((simpleType | complexType)?, (unique | key | keyref)*))
</element>

3.5.6.1 Global elements

$$\left[\text{<element name=NCName1 type=QName>}\right]_{\text{sort\_decl(prefix)}}$$
$$==$$
$$\text{define element } \left[\text{NCName1}\right]_{\text{name(prefix)}} \{ \left[\text{QName}\right]_{\text{name(prefix)}} \}$$

$$\left[\right.$$

<element name=NCName>
Content
</element>

$$\left.\right]_{\text{sort\_decl(prefix)}}$$
$$==$$
$$\text{define element } \left[\text{NCName}\right]_{\text{name(prefix)}} \{ \left[\text{Content}\right]_{\text{group(prefix)}} \}$$

3.5.6.2 Local elements

Local elements with explicit bounds:

$$\left[\text{<element ref=NCName Bound>}\right]_{\text{group(prefix)}}$$
$$==$$
$$\text{element } \left[\text{NCName}\right]_{\text{name(prefix)}} \left[\text{Bound}\right]_{\text{bound}}$$

$$\left[\text{<element name=NCName type=QName Bound>}\right]_{\text{group(prefix)}}$$
$$==$$

$$\text{element } [\text{NCName}]_{\text{name(prefix)}} \{ [\text{QName}]_{\text{name(prefix)}} \} [\text{Bound}]_{\text{bound}}$$

$$[$$

```
<element name=NCName Bound>
   Content
</element>
```

$$]_{\text{group(prefix)}}$$
$$==$$
$$\text{element } [\text{NCName}]_{\text{name(prefix)}} \{ [\text{Content}]_{\text{group(prefix)}} \} [\text{Bound}]_{\text{bound}}$$

Local elements with no explicit bounds:

$$[\text{<element ref=NCName>}]_{\text{group(prefix)}}$$
$$==$$
$$\text{element } [\text{NCName}]_{\text{name(prefix)}}$$

$$[\text{<element name=NCName type=QName>}]_{\text{group(prefix)}}$$
$$==$$
$$\text{element } [\text{NCName}]_{\text{name(prefix)}} \{ [\text{QName}]_{\text{name(prefix)}} \}$$

$$[$$

```
<element name=NCName>
   Content
</element>
```

$$]_{\text{group(prefix)}}$$
$$==$$
$$\text{element } [\text{NCName}]_{\text{name(prefix)}} \{ [\text{Content}]_{\text{group(prefix)}} \}$$

3.5.7 Attributes

The following structure describes attribute declarations in XML Schema.

<attribute

default = string
fixed = string
form = (qualified | unqualified)
id = ID
name = NCName
ref = QName
type = QName
Use
{any attributes with non-schema namespace . . .}>
Content: (annotation?, (simpleType?))
</attribute>

3.5.7.1 Use

The "use" attribute is used to indicate whether attributes are optional or not. The following production describes the structure of the "use"attribute.

The meaning of the "use" attribute is naturally captured in the type system by an occurrence indicator, as reflected in the following mapping rules.

Optional attributes are mapped to optional groups in the type system, using the '?' notation.

$$[\text{optional}]_{\text{use}} == ?$$

Required attributes indicate a required group.

$$[\text{required}]_{\text{use}} == ????????????$$

It is an open issue what should be the semantics of a prohibited attribute in the type system.

$$[\text{prohibited}]_{\text{use}} ==$$

3.5.7.2 Global attributes

$$[\text{<attribute name=NCName type=QName>}]_{\text{sort\_decl(prefix)}} ==$$

$$\text{define attribute } [\![\text{NCName}]\!]_{\text{name(prefix)}} \{ [\![\text{QName}]\!]_{\text{name(prefix)}} \}$$

$$[\![$$

```
<attribute name=NCName>
   Content
</element>
```

$$]\!]_{\text{sort\_decl(prefix)}}$$

$$==$$

$$\text{define attribute } [\![\text{NCName}]\!]_{\text{name(prefix)}} \{ [\![\text{Content}]\!]_{\text{group(prefix)}} \}$$

3.5.7.3 Local attributes

Local attributes with explicit use:

$$[\![\text{<attribute ref=NCName Use>}]\!]_{\text{group(prefix)}}$$

$$==$$

$$\text{attribute } [\![\text{NCName}]\!]_{\text{name(prefix)}} [\![\text{Use}]\!]_{\text{use}}$$

$$[\![\text{<attribute name=NCName type=QName Use>}]\!]_{\text{group(prefix)}}$$

$$==$$

$$\text{attribute } [\![\text{NCName}]\!]_{\text{name(prefix)}} \{ [\![\text{QName}]\!]_{\text{name(prefix)}} \} [\![\text{Use}]\!]_{\text{use}}$$

$$[\![$$

```
<attribute name=NCName Use>
   Content
</attribute>
```

$$]\!]_{\text{group(prefix)}}$$

$$==$$

$$\text{attribute } [\text{NCName}]_{\text{name(prefix)}} \{ [\text{Content}]_{\text{group(prefix)}} \} [\text{Use}]_{\text{use}}$$

Local attributes with no explicit use:

$$[\text{<attribute ref=NCName>}]_{\text{group(prefix)}} == \text{attribute } [\text{NCName}]_{\text{name(prefix)}}?$$

$$[\text{<attribute name=NCName type=QName>}]_{\text{group(prefix)}}? == \text{attribute } [\text{NCName}]_{\text{name(prefix)}} \{ [\text{QName}]_{\text{name(prefix)}} \}?$$

$$[\text{<attribute name=NCName>} \; \text{Content} \; \text{</attribute>}]_{\text{group(prefix)}} == \text{attribute } [\text{NCName}]_{\text{name(prefix)}} \{ [\text{Content}]_{\text{group(prefix)}} \}?$$

3.5.8 Simple Content

For simple content, one just map their content in the type system. The mapping ignores most of the other information, notably facets.

$$[$$

```
<simpleContent
   id = ID
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (restriction | extension))
</simpleContent>
```

$$]_{group(prefix)}$$

$$==$$

$$[\text{Content}]_{group(prefix)}$$

The only part of the <restriction> content that is mapped is simpleType? and the attributes.

$$[$$

```
<restriction
   base = QName
   id = ID
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (simpleType?, (minExclusive | minInclusive | maxExclusive | maxInclusive |
   totalDigits | fractionDigits | length | minLength | maxLength | enumeration | whiteSpace | pattern)*)?,
   ((attribute | attributeGroup)*, anyAttribute?))
</restriction>
```

$$]_{group(prefix)}$$

$$==$$

$$[\text{Content}]_{group(prefix)}$$

All facets are ignored:

$$[$$

(minExclusive | minInclusive | maxExclusive | maxInclusive |
totalDigits | fractionDigits | length | minLength |
maxLength | enumeration | whiteSpace | pattern)*)?

$$]_{group(prefix)}$$

==

We also ignore <extension> for now.

$$[$$

<extension
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((attribute | attributeGroup)*, anyAttribute?))
</extension>

$$]_{group(prefix)}$$
==

3.5.9 Attribute Group

3.5.9.1 Attribute group declaration

$$[$$

<attributeGroup
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((attribute | attributeGroup)*, anyAttribute?))
</attributeGroup>

$$\left]_{\text{sort\_decl(prefix)}} = \text{define type } \left[\text{NCName}\right]_{\text{name(prefix)}} \{ \left[\text{Content}\right]_{\text{group(prefix)}} \}$$

3.5.9.2 Attribute group reference

Attribute group reference can only occur in a complex type:

$$\left[\quad \begin{array}{l} \text{<attributeGroup} \\ \quad \text{id = ID} \\ \quad \text{ref = QName} \\ \quad \text{\{any attributes with non-schema namespace . . .\}>} \\ \quad \text{Content: (annotation?)} \\ \text{</attributeGroup>} \end{array} \right]_{\text{group(prefix)}} = \left[\text{QName}\right]_{\text{name(prefix)}}$$

3.5.9.3 Attribute wildcard

The anyAttribute schema component is mapped to the corresponding wildcard in the XQuery type system. The 'namespace' modifier is ignored.

$$\left[\quad \begin{array}{l} \text{<anyAttribute} \\ \quad \text{id = ID} \\ \quad \textit{namespace = ((##any | ##other) | List of (anyURI | (##targetNamespace | ##local))) : ##any} \\ \quad \textit{processContents = (lax | skip | strict) : strict} \\ \quad \text{\{any attributes with non-schema namespace . . .\}>} \\ \quad \text{Content: (annotation?)} \\ \text{</anyAttribute>} \end{array} \right] = \text{xs:AnyAttribute}$$

3.5.10 Complex Content

$$[$$

```
<complexContent
   id = ID
   mixed = false
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (restriction | extension))
</complexContent>
```

$$]_{group(prefix)} == [\text{Content}]_{group(prefix)}$$

$$[$$

```
<complexContent
   id = ID
   mixed = boolean
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (restriction | extension))
</complexContent>
```

$$]_{group(prefix)} == \text{TEXT}^* \;\&\; [\text{Content}]_{group(prefix)}$$

$$[$$

```
<restriction
   base = QName
   id = ID
   {any attributes with non-schema namespace . . .}>
   Content
</restriction>
```

$$]_{group(prefix)} == [\text{Content}]_{group(prefix)}$$

3.5.11 Simple Types

Global simple types:

$$\left[\right.$$

```
<simpleType
   final = (#all | (list | union | restriction))
   id = ID
   name = NCName
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (restriction | list | union))
</simpleType>
```

$$\left.\right] == \text{define type } \left[\text{NCName}\right]_{\text{group(prefix)}} \{ \left[\text{Content}\right]_{\text{group(prefix)}} \}$$

Anonymous local simple type:

$$\left[\right.$$

```
<simpleType
   final = (#all | (list | union | restriction))
   id = ID
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (restriction | list | union))
</simpleType>
```

$$\left.\right]_{\text{group(prefix)}} == \left[\text{Content}\right]_{\text{group(prefix)}}$$

Again, we ignore all facets and only translate simpleType?

$$[$$

```
<restriction
   base = QName
   id = ID
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (simpleType?, (minExclusive | minInclusive | maxExclusive | maxInclusive |
   totalDigits | fractionDigits | length | minLength | maxLength | enumeration | whiteSpace | pattern)*))
</restriction>
```

$$]_{group(prefix)} == [\text{Content}]_{group(prefix)}$$

$$[$$

```
<list
   id = ID
   itemType = QName
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (simpleType?))
</list>
```

$$] == [\text{Content}]_{group(prefix)}^{*}$$

$$[$$

```
<union
   id = ID
   memberTypes = QName_1, ..., QName_n
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, simpleType_1, ... simpleType_n)
</union>
```

$$] ==$$

$$\left[\text{QName}_1\right]_{\text{name(prefix)}} \mid \ldots \mid \left[\text{QName}_n\right]_{\text{name(prefix)}} \mid \left[\text{simpleType}_1\right]_{\text{group(prefix)}} \mid \ldots \mid \left[\text{simpleType}_n\right]_{\text{group(prefix)}}$$

3.6 Major type issues

Ed. Note: Alignment between XQuery, the [XQuery 1.0 and XPath 2.0 Data Model], and the XQuery formal semantics. There are some known discrepancies between the type models in the Data Model document, the XQuery document and this document. Currently the XQuery/XPath grammar does not contain comments, processing-instructions or schema-components. Currently neither XQuery nor this document contain text nodes or document nodes, which are present in the Data Model. See issues **[Issue-0068: Document Collections]**, **[Issue-0105: Types for nodes in the data model.]**.

Ed. Note: Syntax for types. With respect to the currently published XQuery document, there is another major misalignment as well: the current XQuery document does not have a syntax for type constructors in XQuery.

Ed. Note: Complexity of type operations. There are concerns about the complexity of type inference, notably about type subsumption used during type inference, and validation used during evaluation of some of the type operations (e.g., typeswitch). Additionally, there are many language features for which it is possible to define special type inference rules that would give tighter bounds. The question is, which features, which rules, and how do they interact with the complexity of type inference and subtype computation. See **[Issue-0080: Typing of parent]**, **[Issue-0083: Expressive power and complexity of typeswitch expression]**, **[Issue-0091: Attribute expression]**.

Ed. Note: XML Schema mapping. The mapping from XML Schema to the XQuery type system is new and has not yet been reviewed by the XML Query Working Group See **[Issue-0019: Support derived types]**, **[Issue-0020: Structural vs. name equivalence]**, **[Issue-0073: Facets for simple types and their role for typechecking]**.

4 Semantics of Expressions

Ed. Note: Status: This section contained a fully revised semantics for XQuery, based on the December 2001 working drafts. Some sections might still await further material or editorial consolidation, as indicated by additional status note included in each specific section.

This section defines semantics of all XQuery expressions. The organization of this section parallels the organization of Section 2 of the XQuery document.

[4] Expr ::= SortExpr
| OrExpr
| AndExpr
| FLWRExpr
| QuantifiedExpr
| TypeswitchExpr
| IfExpr
| GeneralComp
| ValueComp
| NodeComp
| OrderComp
| InstanceofExpr
| RangeExpr
| AdditiveExpr
| MultiplicativeExpr
| UnionExpr
| IntersectExceptExpr
| UnaryExpr
| CastExpr
| Constructor
| PathExpr

For each expression, we give a short description of the expression to be defined and include its grammar productions. The semantics of an expression includes the normalization, static analysis, and dynamic evaluation phases. Recall that normalization rules translate XQuery syntax into core XQuery syntax. In the sections that contain normalization rules, we include the Core grammar productions into which the expression is normalized. After normalization, sections on static type inference and dynamic evaluation define the static type and dynamic value for the core expression.

The different phases are covered by the judgments shown in the following table where the environment patterns are defined as part of the context below.

statEnvs |- *query phrase*	The *query phrase* is well-typed in the static environment group statEnvs, **[2.5.5 Static type inference]**.
statEnvs |- *query phrase* : *type expression*	The *query phrase* is well-typed in the static environment group statEnvs, and its static type is given by *type expression*, **[2.5.5 Static type inference]**.
dynEnvs |- *query phrase* => *value phrase*	The evaluation of *query phrase* yields the value given by *value phrase* in the dynamic environment group dynEnvs, **[2.5.6 Evaluation rules]**

4.1 Basics

4.1.1 Expression Context

Introduction

The expression context for a given expression consists of all the information that can affect the result of the expression. This information is organized into the *static context* and the *evaluation context*. This section specifies the environments that represent the context information used by XQuery expressions.

4.1.1.1 Static Context

The XQuery static inference rules use the "static" environment group **statEnvs** containing the environments built during static type checking and used during both static type checking and dynamic evaluation.

The following environments are maintained in the static environment group:

statEnvs.namespace	The *namespace* environment maps namespace prefixes (*NCName*s) onto their fully qualified definitions (an xs:anyURI), as determined by the appropriate namespace declaration.
statEnvs.varType	The *static variable type* environment maps variable and parameter names (*Variable*s) to their static type (a *Type*).
statEnvs.funcType	The *function declaration* environment stores the static type signatures of functions. Because XQuery allows multiple functions with the same name differing only in the number of arguments, this environment maps function name/arity (*QName*, integer) pairs to function type signatures ($Type_r$, $Type_1$, ..., $Type_n$) where the first is the return type and the rest are the types of the parameters, in order.
statEnvs.elemDecl	The *element declaration* environment maps element type names (*QName*s) onto their element type definition (a *Type*).
statEnvs.attrDecl	The *attribute declaration* environment maps attribute type names (*QName*s) onto their attribute type definition (a *Type*).
statEnvs.typeDecl	The *type declaration* environment maps type names (*QName*s) onto their type definition (a *Type*).

Environments have an initial state when query processing begins, containing, for example, the function signatures of all built-in functions.

A common use of the static environment is to expand *QName*s by looking up namespace prefixes in the statEnvs.namespace environment. The *QName* production in the XQuery grammar is as follows:

[223] QName ::= ":"? NCName (":" NCName)?

which corresponds to either a single local name, or a namespace prefix and a local name.

We define a helper function, `expand` to expand *QName*s by looking up the namespace prefix in statEnvs.namespace. We write it as follows:

statEnvs |- expand(*QName*) = *qname*(*anyURI*, *ncname*)

or

statEnvs |- expand(*QName*) = *qname*(*ncname*)

where what is on the right side is a QName *value* (not a QName expression).

The helper expand function is defined as follows:

- If *QName* matches $NCName_1$: $NCName_2$, and if statEnvs.namespace($NCName_1$) = *anyURI*, then the expression yields *qname*(*anyURI*,$NCName_2$). If the namespace prefix $NCName_1$ is not found in statEnvs.namespace, then the expression does not apply (that is, the inference rule will not match).
- If *QName* matches *NCName*, and if statEnvs.namespace("*default*") = *anyURI*, then the expression yields *qname*(*anyURI*,*NCName*), where *anyURI* is the default namespace in effect, otherwise the expression yields *qname*(*NCName*).

  Ed. Note: The above rules could be given as proper inference rules defining the expand *judgment*.

Here is an example that shows modifying a static environment in response to a namespace definition.

$$\frac{\text{statEnvs [namespace}(NCName \text{ |-> } StringLiteral) \text{] |- } Expr^*}{\text{statEnvs |- namespace } NCName = StringLiteral\ Expr^*}$$

This rule is read: "the phrase on the bottom (a namespace declaration followed by a sequence of expressions) is well-typed (accepted by the static type inference rules) within an environment statEnvs *if* (above the line, now) the sequence of expressions is well-typed in the environment obtained from statEnvs by adding the namespace declaration".

This is the common idiom for passing new information in an environment along to sub-expressions. In the case where we need to update an environment with a completely new component, we write:

statEnvs [namespace = (*NewEnvironment*)]

4.1.1.2 Evaluation Context

We use **dynEnv** to denote the group of "dynamic" environments, i.e., those environments built and used only during dynamic evaluation.

The following environments are dynamic:

dynEnvs.funcDefn	The *dynamic function* environment maps a function name to a function definition. The function definition consists of an expression, which is the function's body, and a list of variables, which are the function's formal parameters. As with statEnvs.funcType, dynEnvs.funcDefn requires a function name and a parameter count: it maps a (*QName*, integer) pair to a value list of the form (*Expr* , $Variable_1$, ..., $Variable_n$).

dynEnvs.varValue The *dynamic value* environment maps a variable name (*QName*) onto the variables current value (*value*).

Ed. Note: DD: this still does not account for those functions that are genuinely overloaded, as some of the built-in functions are. Probably this will be handled by special rules for those functions in the main semantics section of the document. See **[Issue-0122: Overloaded functions]**

Finally, dynamic environments have an initial state when query processing begins, containing, for example, the function declarations of all built-in functions.

Ed. Note: Somewhere we need an exact definition of what is in the initial environments? Notice that for XPath this is partially defined by the containing language. See **[Issue-0115: What is in the default context?]**.

Several built-in variables represent other parts of the evaluation context:

Built-in Variable	Represents:
$fs:dot	context item
$fs:position	context position
$fs:last	context size
	context document

Variables with the "xq" namespace prefix are reserved for use in the definition of the formal semantics. It is a static error to define a variable in the "xq" namespace.

Ed. Note: "xq" is actually a namespace *prefix* and should be replaced with a proper namespace URI unique to this spec. See **[Issue-0100: Namespace resolution]**.

Ed. Note: The following dynamic contexts have no formal representation yet: current date and time. See **[Issue-0114: Dynamic context for current date and time]**

4.1.2 Type Conversions

Notation

For the basic-conversion rules, we define three normalization functions: $[]_{\text{Optional\_Atomic\_Value}}$ $[]_{\text{Atomic\_Value\_Sequence}}$, and $[]_{\text{Node\_Sequence}}$. They are used in the definition of arithmetic, comparison, and function-call expressions.

Normalization

There are two variants of the normalization function $[\,]_{\text{Optional\_Atomic\_Value}}$: the first variant is not parameterized by the required type of the expression and the second variant is parameterized by the required type. The function converts an expression into an expression that returns an optional atomic value and is used in the normalization of expressions whose required type is an optional atomic value. A node is converted to an optional atomic value by extracting its typed content.

$$[\,\textit{Expr}\,]_{\text{Optional\_Atomic\_Value}} \;==$$

```
typeswitch (Expr) as $v
case fs:atomic? return $v
case node return
  (typeswitch (fs:data($v)) as $w
    case fs:atomic? return $w
    default return [$w]Type_Exception_Opt_Atomic)
default return [$v]Type_Exception_Other
```

In the parameterized variant, an atomic value with type fs:UnknownSimpleType is always cast to the required type:

$$[\,\textit{Expr}\,]_{\text{Optional\_Atomic\_Value(Type)}} \;==$$

```
typeswitch (Expr) as $v
case fs:UnknownSimpleType return cast as Type ($v)
case fs:atomic? return $v
case node return
  (typeswitch (fs:data($v)) as $w
    case fs:UnknownSimpleType return cast as Type ($w)
    case fs:atomic? return $w
    default return [$w]Type_Exception_Opt_Atomic(Type))
default return [$v]Type_Exception_Opt_Atomic(Type)
```

Ed. Note: The type fs:UnknownSimpleType denotes untyped character data. The name of this type is an open XPath/XQuery issue (Issue-174).

The function $[\,]_{\text{Atomic\_Value\_Sequence(Type)}}$ is parameterized by the atomic type of the sequence. This rule is used in the normalization of expressions whose required type is a sequence of atomic values. The rule converts each value in the argument expression to an atomic value. An atomic value with type fs:UnknownSimpleType is always cast to the required type. A node is converted to a sequence of atomic values by extracted its typed content and, for each atomic value in that sequence, cast any value with type fs:UnknownSimpleType to the required type and return all other values unchanged.

$$[\![\mathit{Expr}]\!]_{\text{Atomic\_Value\_Sequence(Type)}}$$
==
for $v in *Expr*
return
  typeswitch ($v) as $w
  case fs:UnknownSimpleType return cast as *Type* ($w)
  case atomic return $w
  case node return for $x in fs:data($w) return $[\![\$x]\!]_{\text{Optional\_Atomic\_Value(Type)}}$
  default return $[\![\$v]\!]_{\text{Type\_Exception\_Other}}$

The function $[\,]_{\text{Node\_Sequence}}$ is used in the normalization of expressions whose required type is node sequence. They all map an XQuery core expression to an XQuery core expression.

$$[\![\mathit{Expr}]\!]_{\text{Node\_Sequence}}$$
==
typeswitch (*Expr*) as $v
case node* return $v
default return $[\![\$v]\!]_{\text{Type\_Exception\_Other}}$

Notation

Two more normalization functions implement type exceptions They are: $[\,]_{\text{Type\_Exception\_Opt\_Atomic(Type)}}$ and $[\,]_{\text{Type\_Exception\_Other}}$.

Normalization

These definitions implement the `strict` type exception policy. The strict policy always raises an error:

$$[\![Expr]\!]_{\text{Type\_Exception\_Opt\_Atomic(Type)}}$$

==

dm:error()

$$[\![Expr]\!]_{\text{Type\_Exception\_Other}}$$

==

dm:error()

In the first rule, the required type is xs:boolean. The conditional expression in the "default return" clause below guarantees that in an arbitrary sequence of items, if at least one value is a node, then the boolean expression evaluates to true.

$$[\![Expr]\!]_{\text{Type\_Exception\_Opt\_Atomic(boolean)}} = \cdots$$

==

typeswitch (*Expr*) as $v
case () return xf:false()
default return
  if (xf:length($v/self::node()$_{\text{Path}}$) >= 1) then xf:true()
  else xf:boolean(xf:item-at($v, 1))

Ed. Note: MFF: The rule above requires that $v/self::node() on a simple value returns () rather than raise an error.

Ed. Note: The semantics of Boolean node tests over sequences is still an open issue. See [**Issue-0131: Boolean node test and sequences**].

In the next rule, the required type is node type. The conditional expression in the "default return" clause below guarantees that in an arbitrary sequence of items, if the *first* item is a node, then the boolean expression evaluates to true.

$$[\![Expr]\!]_{\text{Type\_Exception\_Opt\_Atomic(node)}} = \cdots$$

==

typeswitch (*Expr*) as $v
case item+ return
    (typeswitch (xf:item-at($v, 1)) as $w
    case node return $w
    default return dm:error())
default return dm:error()

Ed. Note: MFF: All the remaining rules in the fallback conversions table must be specified. See

[Issue-0113: Incomplete specification of type conversions]

$$[\![Expr]\!]_{\text{Type\_Exception\_Other}} == \text{dm:error()}$$

4.2 Primary Expressions

This section defines the semantics of [2.2 Primary Expressions] in [XQuery 1.0: A Query Language for XML]. Primary expressions are the basic primitives of the language.

A primary expression is a variable, literal, an element name, a function call, a wildcard expression, a node-kind test, or a parenthesized expression.

[47]	PrimaryExpr	::=	Variable | Literal | ElementNameOrFunctionCall | Wildcard | KindTest | ParenthesizedExpr

4.2.1 Literals

Introduction

A **literal** is a direct syntactic representation of a atomic value. XQuery supports two kinds of literals: string literals and numeric literals.

[49]	Literal	::=	NumericLiteral | StringLiteral
[48]	NumericLiteral	::=	IntegerLiteral | DecimalLiteral | DoubleLiteral
[201]	IntegerLiteral	::=	Digits
[202]	DecimalLiteral	::=	("." Digits) | (Digits "." [0-9]*)
[203]	DoubleLiteral	::=	(("." Digits) | (Digits ("." [0-9]*)?)) ([e] | [E]) ([+] | [-])? Digits
[214]	StringLiteral	::=	(["] [^"]* ["]) | (['] [^']* ['])

Normalization

All literals are core expressions, therefore there are no normalization rules for literals.

Core Grammar

The core grammar rules for literals are:

[49]	Literal	::=	NumericLiteral | StringLiteral
[48]	NumericLiteral	::=	IntegerLiteral | DecimalLiteral | DoubleLiteral

Static Type Analysis

In the static semantics, the type of an integer literal is simply xs:integer:

statEnvs |- IntegerLiteral : xs:integer

Dynamic Evaluation

In the dynamic semantics, an integer literal is evaluated by constructing a atomic value in the data model, which consists of the literal value and its type:

dynEnvs |- IntegerLiteral => dm:atomic-value(IntegerLiteral, xs:integer)

The formal definitions of decimal, double, and string literals are analogous to those for integer.

Static Type Analysis

statEnvs |- DecimalLiteral : xs:decimal

Dynamic Evaluation

dynEnvs |- DecimalLiteral => dm:atomic-value(DecimalLiteral, xs:decimal)

Static Type Analysis

statEnvs |- DoubleLiteral : xs:double

Dynamic Evaluation

dynEnvs |- DoubleLiteral => dm:atomic-value(DoubleLiteral, xs:double)

Static Type Analysis

statEnvs |- StringLiteral : xs:string

Dynamic Evaluation

dynEnvs |- StringLiteral => dm:atomic-value(StringLiteral, xs:string)

> **Ed. Note:** MFF: Phil has noted that the data model should support primitive literals in their lexical form, in which case no explicit dynamic semantic rule would be necessary. See [**Issue-0118: Data model syntax and literal values**].

4.2.2 Variables

Introduction

A **variable** evaluates to the value to which the variable's QName is bound in the **evaluation context**.

[192] Variable ::= "$" QName

Normalization

A variable is a core expression, therefore there is no normalization rule for a variable.

Core Grammar

The core grammar rules for variables are:

[192] Variable ::= "$" QName

Static Type Analysis

In the static semantics, the type of a variable is simply its type in the static type environment statEnvs.varType:

$$\frac{\text{statEnvs.varType}(\textit{Variable}) = \textit{Type}}{\text{statEnvs} \mid\!\!- \textit{Variable} : \textit{Type}}$$

In case the variable is not bound in the environment, the system raises an error.

Dynamic Evaluation

In the dynamic semantics, a variable is evaluated by "looking up" its value in dynEnvs.varValue:

$$\frac{\text{dynEnvs.varValue}(\textit{Variable}) = \textit{value}}{\text{dynEnvs} \mid\!\!- \textit{Variable} \Rightarrow \textit{value}}$$

In case the variable is not bound in the environment, the system raises an error.

4.2.3 Parenthesized Expressions

The formal definition of parenthesized expressions is in **[4.4 Sequence Expressions]**.

4.2.4 Function Calls

Introduction

A function call consists of a QName followed by a parenthesized list of zero or more expressions.

[51] ElementNameOrFunctionCall ::= QName ("(" (Expr ("," Expr)*)? ")")?

Notation

We define a normalization function, $[\,]_{\text{Formal\_Argument}}$ for mapping the formal argument of a function call. There are three variants of this mapping rule. If the required type of the formal argument is an optional atomic value, the following rule is applied: (*Type* is the required atomic type.)

$$[\,Expr\,]_{\text{Formal\_Argument}} \;==\; [\,Expr\,]_{\text{Optional\_Atomic\_Value(Type)}}$$

If the required type of the formal argument is a sequence of atomic values, the following rule is applied: (*Type* is the atomic type of the sequence.)

$$[\,Expr\,]_{\text{Formal\_Argument}} \;==\; [\,Expr\,]_{\text{Atomic\_Value\_Sequence(Type)}}$$

If the required type of the formal argument is neither an optional atomic value or a sequence of atomic values, the expression is simply mapped to its corresponding core expression:

$$[\,Expr\,]_{\text{Formal\_Argument}} \;==\; [\,Expr\,]$$

Normalization

First, each argument in a function call is normalized to its corresponding core expression by applying $[\,]_{\text{Formal\_Argument}}$.

$$[\; QName\ (Expr_1, ..., Expr_n)\;] \;==\; [\,QName\,]\ (\,[\,Expr_1\,]_{\text{Formal\_Argument}}, ..., [\,Expr_n\,]_{\text{Formal\_Argument}})$$

Note that this normalization rule depends on the static environment containing function signatures. This is working as the static context is built from the query prolog, before the normalization phase. **[5 The Query Prolog]** explains how the query prolog is processed.

Core Grammar

The core grammar rule for a function call is:

[51] ElementNameOrFunctionCall ::= QName "(" (Expr ("," Expr)*)? ")"

Static Type Analysis

In the static semantics for core function calls, the type of each actual argument to the function must be a subtype of the corresponding formal argument to the function, i.e., it is not necessary that the actual and formal types be equal.

$$\frac{\begin{array}{c}\text{statEnvs} \vdash \mathit{QName} \Rightarrow \mathit{qname} \\ \text{statEnvs.funcType}(\mathit{qname}, n) = \mathit{qname}(\mathit{Type}_1, ..., \mathit{Type}_n) \text{ returns } \mathit{Type} \\ \text{statEnvs} \vdash \mathit{Expr}_1 : \mathit{Type'}_1 \quad \mathit{Type'}_1 <: \mathit{Type}_1 \\ ... \\ \text{statEnvs} \vdash \mathit{Expr}_n : \mathit{Type'}_n \quad \mathit{Type'}_n <: \mathit{Type}_n \end{array}}{\text{statEnvs} \vdash \mathit{QName}\ (\mathit{Expr}_1, ..., \mathit{Expr}_n) : \mathit{Type}}$$

Dynamic Evaluation

In the dynamic semantics for core function calls, we first evaluate each function argument. Then we extend dynEnvs.varValue by binding each formal variable to its corresponding actual value, and then evaluate the body of the function in the new environment. The resulting value is the value of the function call.

$$\frac{\begin{array}{c}\text{dynEnvs} \vdash \mathit{QName} \Rightarrow \mathit{qname} \\ \text{dynEnvs.funcDefn}(\mathit{qname}, n) = (\mathit{Expr}, \mathit{Variable}_1, ..., \mathit{Variable}_n) \\ \text{dynEnvs} \vdash \mathit{Expr}_1 \Rightarrow \mathit{value}_1 \ ... \ \text{dynEnvs} \vdash \mathit{Expr}_n \Rightarrow \mathit{value}_n \\ \text{dynEnvs}\ [\ \text{varValue} = (\mathit{Variable}_1 \mapsto \mathit{value}_1; ...; \mathit{Variable}_n \mapsto \mathit{value}_n)\ ] \vdash \mathit{Expr} \Rightarrow \mathit{value} \end{array}}{\text{dynEnvs} \vdash \mathit{QName}\ (\ \mathit{Expr}_1, ..., \mathit{Expr}_n\ ) \Rightarrow \mathit{value}}$$

Note that the function body is evaluated in the *default* environment,

> **Ed. Note:** This semantics only works for non-overloaded function. This is always the case for user defined functions, but not for [XQuery 1.0 and XPath 2.0 Functions and Operators] functions. See **[Issue-0122: Overloaded functions]**.

4.2.5 Comments

[106] ExprComment ::= "{--" [^}]* "--}"

Comments are lexical constructs only, and have no meaning within the query and therefore have no formal semantics.

4.3 Path expressions

Introduction

Path expressions are used to locate nodes within a tree. There are two kinds of path expressions, absolute path expressions and relative path expressions. An absolute path expression is a rooted relative path expression. A relative path expression is composed of a sequence of steps.

[25] PathExpr ::= AbsolutePathExpr | RelativePathExpr
[31] AbsolutePathExpr ::= ("/" RelativePathExpr?) | ("//" RelativePathExpr)

[32] RelativePathExpr ::= StepExpr (("/" | "//") StepExpr)*
[33] StepExpr ::= AxisStep | GeneralStep

Notation

Besides the general $[\text{Expr}]_{\text{Expr}}$ normalization rule for expressions, we use an auxiliary normalization rule for path expressions, denoted $[\text{PathExpr}]_{\text{Path}}$.

Normalization

A path expression always returns a sequence in document order. The complete normalization of a Path expression is obtained by applying the Path normalization and then sorting the result by document order and removing duplicates.

$$[\text{PathExpr}]_{\text{Expr}} == \text{fs:distinct-doc-order}([\text{PathExpr}]_{\text{Path}})$$

> **Ed. Note:** Jerome: the restriction that XPath expressions operate on nodes here seems too strict and is still an open issue. See [**Issue-0125: Operations on node only in XPath**].

Absolute path expressions are path expressions starting with the `/` symbol, indicating that the expression must be applied on the root node in the current context. Remember that the root node in the current context is the topmost ancestor of the context node. The following two rules are used to normalize absolute path expressions, and rely on the use of the `fs:document()` function, which computes the document node from the context node.

$$[\text{"/"}]_{\text{Path}} == [\text{xf:document(\$fs:dot)}]_{\text{Path}}$$

$$[\text{"/" RelativePathExpr}]_{\text{Path}} == [\text{xf:document(\$fs:dot) "/" RelativePathExpr}]_{\text{Path}}$$

> **Ed. Note:** Some of the semantics of the root expression `'/'` is still an open issue. For instance, what should be the semantics of `'/'` in case of a document fragment (e.g., created using XQuery element constructor). See [**Issue-0123: Semantics of /**].

The `//` syntax is part of the Abbreviated syntax in XPath. We give separate normalization rules for Abbreviated XPath expressions in **[4.3.5 Abbreviated Syntax]**

Normalization of relative path expressions is done by normalizing each step, one after the other. A Step is

normalized by using a `for` expression in the following way.

$$\left[\text{StepExpr "/" RelativePathExpr}\right]_{\text{Path}} \;==\; \text{let \$fs:sequence := fs:distinct-doc-order(} \left[\text{StepExpr}\right]_{\text{Path}} \text{) return let \$fs:last := xf:length(\$fs:sequence) return for \$fs:dot in \$fs:sequence return let \$fs:position := xf:index-of(\$fs:sequence, \$fs:dot) return } \left[\text{RelativePathExpr}\right]_{\text{Path}}$$

As explained in the XQuery document, applying a step in XPath changes the focus (or context). The change of focus is made explicit by the normalization rule, which binds the variable `$fs:dot` to the node currently being processed, and the variable `$fs:position` to the position (i.e., the position within the input sequence) of that node.

> **Ed. Note:** This rule is using the function xf:index-of to bind the position of a node in a sequence. This is working since path expressions should only work on nodes. See **[Issue-0125: Operations on node only in XPath]**. Still, it might be preferable to be able to bind both the current node and its position through a single processing of the collection within the for expression. The FS editors proposed several syntax for such an operation. For instance: `for $v at $i in E1 return E2`. See **[Issue-0124: Binding position in FLWR expressions]**.

The semantics of Step expressions, which are either axis or general steps, is given next.

4.3.1 Axis Steps

Introduction

An axis step is either a combination of an axis accessor, a node test and a step qualifier, or an abbreviated step. We define here the semantics of Axis NodeTest pairs. The semantics of StepQualifiers is defined in **[4.3.3 Step Qualifiers]** and the semantics of abbreviated steps is defined in **[4.3.5 Abbreviated Syntax]**.

[34] AxisStep ::= (Axis NodeTest StepQualifiers) | AbbreviatedStep

> **Ed. Note:** (Michael Kay): there is a bug here in the formal semantics which does not deal properly with principal node types. See **[Issue-0108: Principal node types in XPath]**.

Notation

During the normalization process, we have to distinguish between forward and reverse Axis. In order to do so, we need to slightly reorganize the XQuery grammar production to make that distinction. This reorganization does not actually change the syntax, but merely regroup together different productions. Here is the grammar that is the basis for the following normalization of steps.

[96] FSAxisStep ::= (ForwardAxis NodeTest StepQualifiers)
| (ReverseAxis NodeTest StepQualifiers)
| AbbreviatedStep

[97]	ForwardAxis	::=	"self" "::" | "child" "::" | "descendant" "::" | "descendant-or-self" "::" | "attribute" "::"
[98]	ReverseAxis	::=	"parent" "::"

We then use two auxiliary normalization rules denoted $[\ldots]_{\text{ForwardPath}}$ and $[\ldots]_{\text{ReversePath}}$

Normalization

A forward axis is directly translated to core XQuery

$$[\text{ForwardAxis NodeTest StepQualifier}]_{\text{Path}} == [\text{ForwardAxis NodeTest StepQualifier}]_{\text{ForwardPath}}$$

A reverse axis is directly translated to core XQuery

$$[\text{ReverseAxis NodeTest StepQualifier}]_{\text{Path}} == [\text{ReverseAxis NodeTest StepQualifier}]_{\text{ReversePath}}$$

Core Grammar

We have now normalized all Axis Steps into the following core grammar.

[34]	AxisStep	::=	Axis NodeTest
[35]	Axis	::=	"child" "::" | "descendant" "::" | "parent" "::" | "attribute" "::" | "self" "::" | "descendant-or-self" "::"
[36]	NodeTest	::=	NameTest | KindTest
[37]	NameTest	::=	QName | Wildcard
[38]	Wildcard	::=	"*" | ":"? NCName ":" "*" | "*" ":" NCName
[39]	KindTest	::=	ProcessingInstructionTest | CommentTest | TextTest | AnyKindTest
[40]	ProcessingInstructionTest	::=	"processing-instruction" "(" StringLiteral? ")"
[41]	CommentTest	::=	"comment" "(" ")"
[42]	TextTest	::=	"text" "(" ")"

[43] AnyKindTest ::= "node" "(" ")"

Next, we give the static and dynamic semantics for Axis Steps.

Notation

We use the term *Filter* to denote either an axis or a node test.

[95] Filter ::= Axis | NodeTest

To define the semantics of axis and nodetest, we use the two following auxiliary judgments for filters. The first judgment is defining the effect of a filter (either an axis or a nodetest) on a value, and is used in the dynamic semantics. This judgment should be read as follows: in the current context (i.e., with respect to the dynamic environment dynEnvs.varValue), applying the filter `Filter` on value $\texttt{Value}_1$ yields the value $\texttt{Value}_2$

$$\text{dynEnvs.varValue} \ ; \ \text{Value}_1 \ \text{|-} \ \text{Filter} \Rightarrow \text{Value}_2$$

The second judgment is defining the effect of a filter (either an axis or a nodetest) on a type and is used in the static semantics. This judgment should be read as follows: in the current context, (i.e., with respect to the the static environment statEnvs.varType), applying the filter `Filter` on type $\texttt{Type}_1$ yields the type $\texttt{Type}_2$

$$\text{statEnvs.varType} \ ; \ \text{Type}_1 \ \text{|-} \ \text{Filter} : \text{Type}_2$$

In a few cases, when applying the type filter judgment recursively, we need to use an additional type environment that contains local type information. We use the following notation.

$$\text{statEnvs} \ ; \ \text{localTypeEnv} \ ; \ \text{Type}_1 \ \text{|-} \ \text{Filter} : \text{Type}_2$$

We write updates to the local type environment as "localTypeEnv[*qname* : *Type*]"

Static Type Analysis

The static semantics of an Axis NodeTest pair is obtained by retrieving the type of the context node, and applying the two filters (Axis, then NodeTest) on the result. The application of each filter is expressed through the static filter judgment as follows.

$$\frac{\begin{array}{c}\text{statEnvs.varType}(\$\text{fs:dot}) = \text{Type}_1 \\ \text{statEnvs} \ ; \ \text{Type}_1 \ \text{|-} \ \text{Axis} : \text{Type}_2 \\ \text{statEnvs} \ ; \ \text{Type}_2 \ \text{|-} \ \text{NodeTest} : \text{Type}_3\end{array}}{\text{statEnvs} \ \text{|-} \ \text{Axis NodeTest} : \text{Type}_3}$$

Dynamic Evaluation

The dynamic semantics of an Axis NodeTest pair is obtained by retrieving the context node, and applying the two filters (Axis, then NodeTest) on the result. The application of each filter is expressed through the filter judgment as follows.

$$\frac{\begin{array}{c}\text{dynEnvs.varValue}(\$\text{fs:dot}) = \text{Value}_1 \\ \text{dynEnvs.varValue} \ ; \ \text{Value}_1 \ \text{|-} \ \text{Axis} \Rightarrow \text{Value}_2 \\ \text{dynEnvs.varValue} \ ; \ \text{Value}_2 \ \text{|-} \ \text{NodeTest} \Rightarrow \text{Value}_3\end{array}}{\text{dynEnvs.varValue} \ \text{|-} \ \text{Axis NodeTest} \Rightarrow \text{Value}_3}$$

We now define the filter judgment used in the dynamic semantics. This is done in two sets of inference rules. The first set of inference rules is common to all filters, and defines the part of the semantics which applies to both Axis and NodeTest. The second set of inference rules is specific to either Axis or NodeTest and is given in the corresponding subsections below.

Static Type Analysis

Here are the common set of inference rules defining the static semantics of filters. Essentially, these rules are used to process complex types through a filter, breaking the types down to the type of a node or of a value. The semantics of a filter applied to a node or value type is then specific to each Axis and NodeTest.

$$\frac{}{\text{statEnvs} \;;\; () \;|\!-\; \text{Filter} : ()}$$

$$\frac{}{\text{statEnvs} \;;\; \text{none} \;|\!-\; \text{Filter} : \text{none}}$$

$$\frac{\begin{array}{c}\text{statEnvs} \;;\; \text{Type}_1 \;|\!-\; \text{Filter} : \text{Type'}_1 \\ \text{statEnvs} \;;\; \text{type}_2 \;|\!-\; \text{Filter} : \text{Type'}_2\end{array}}{\text{statEnvs} \;;\; \text{Type}_1, \text{Type}_2 \;|\!-\; \text{Filter} : \text{Type'}_1,\text{Type'}_2}$$

$$\frac{\begin{array}{c}\text{statEnvs} \;;\; \text{Type}_1 \;|\!-\; \text{Filter} : \text{Type'}_1 \\ \text{statEnvs} \;;\; \text{type}_2 \;|\!-\; \text{Filter} : \text{Type'}_2\end{array}}{\text{statEnvs} \;;\; \text{Type}_1 \mid \text{Type}_2 \;|\!-\; \text{Filter} : \text{Type'}_1 \mid \text{Type'}_2}$$

$$\frac{\begin{array}{c}\text{statEnvs} \;;\; \text{Type}_1 \;|\!-\; \text{Filter} : \text{Type'}_1 \\ \text{statEnvs} \;;\; \text{type}_2 \;|\!-\; \text{Filter} : \text{Type'}_2\end{array}}{\text{statEnvs} \;;\; \text{Type}_1 \;\&\; \text{Type}_2 \;|\!-\; \text{Filter} : \text{Type'}_1 \;\&\; \text{Type'}_2}$$

Dynamic Evaluation

Here are the common set of inference rules defining the dynamic semantics of filters. These rules apply filters to a sequence, by breaking the sequence down to a node or value. The semantics of a filter applied to a node or value is then specific to each Axis and NodeTest.

$$\frac{}{\text{dynEnvs.varValue} \;;\; () \;|\!-\; \text{Filter} \Rightarrow ()}$$

$$\frac{\begin{array}{c}\text{dynEnvs.varValue} \;;\; \text{Value}_1 \;|\!-\; \text{Filter} \Rightarrow \text{Value'}_1 \\ \text{dynEnvs.varValue} \;;\; \text{Value}_2 \;|\!-\; \text{Filter} \Rightarrow \text{Value'}_2\end{array}}{\text{dynEnvs.varValue} \;;\; \text{Value}_1, \text{Value}_2 \;|\!-\; \text{Filter} \Rightarrow \text{Value'}_1,\text{Value'}_2}$$

Ed. Note: Jerome: These rules use a notation abuse. `Value'`$_1$`,Value'`$_2$ indicates the

concatenation of $Value'_1$ and $Value'_2$, and is used for both constructing a new sequence and deconstructing it. When doing the construction, it is equivalent to applying the `op:concatenate` operator. We should strive for more consistent notation for data model constructions and deconstruction, throughout the formal semantics specification. See also **[Issue-0118: Data model syntax and literal values]**.

4.3.1.1 Axes

Introduction

[35] Axis ::= "child" "::"
| "descendant" "::"
| "parent" "::"
| "attribute" "::"
| "self" "::"
| "descendant-or-self" "::"

Axis are used to traverse the document, returning nodes related to the context node inside the tree (children, parent, attribute, etc).

Static Type Analysis

The following rules define the static semantics of the filter judgment when applied to an Axis.

The type for the self axis is the same type as the type of the context node.

statEnvs ; NodeType |- self:: : NodeType

In case of elements, the type of the child axis is obtained by extracting the children types out of the content model describing the type of the context node.

statEnvs ; element *qname* { AttrType, ChildType } |- child:: : ChildType

The type of the attribute axis is obtained by extracting the attribute types out of the content model describing the type of the context node.

statEnvs ; element *qname* { AttrType, ChildType } |- attribute:: : AttrType

The type for the parent axis is always an element, possibly optional.

statEnvs ; element |- parent:: : element?

Ed. Note: Better typing for the parent axis is still an open issue. See **[Issue-0080: Typing of parent]**.

The type for the namespace axis is always empty.

statEnvs NodeType |- namespace:: : ()

Ed. Note: Jerome: The type of namespace nodes is still an open issue. See [**Issue-0105: Types for nodes in the data model.**].

The types for the descendant, and descendant-or-self axis are implemented through recursive application of the children and parent filters. The corresponding inference rules use the auxiliary filter judgment with a local type environment. This local type environment is used to keep track of the visited elements in order to deal with recursive types.

$$\frac{\text{statEnvs} \; ; \; \{ \} \; ; \; \text{Type} \; |\text{-} \; \text{descendant::} \; : \; \text{Type}'}{\text{statEnvs} \; ; \; \text{Type} \; |\text{-} \; \text{descendant::} \; : \; \text{Type}'}$$

The two following rules are used to deal with global elements. Global elements need careful treatment here in order to deal with possible recursion. Notably, if the global element has been seen before, then the rule terminates and returns the union of all types already visited in the local environment.

$$\frac{\begin{array}{c}\text{localTypeEnv}(\textit{qname}) = \text{Type} \\ \text{localTypeEnv} = \{ \; \text{Type}_1, \text{Type}_2, \ldots \; \}\end{array}}{\text{statEnvs} \; ; \; \text{element} \; \textit{qname} \; |\text{-} \; \text{descendant::} \; : \; (\text{Type}_1 \mid \text{Type}_2 \mid \ldots)^*}$$

$$\frac{\begin{array}{c}\text{localTypeEnv}(\textit{qname}) \text{ is an error} \\ \text{statEnvs.varType}(\textit{qname}) = \text{Type} \\ \text{statEnvs} \; ; \; \text{localTypeEnv}[\textit{qname} : \text{Type}] \; ; \; \text{Type} \; |\text{-} \; \text{descendant::} \; : \; \text{Type}'\end{array}}{\text{statEnvs localTypeEnv} \; ; \; \text{element} \; \textit{qname} \; |\text{-} \; \text{descendant::} \; : \; \text{element} \; \textit{qname}, \; \text{Type}'}$$

Ed. Note: Peter: I need yet to figure out, whether this works, there still appear a few glitches. See [**Issue-0132: Typing for descendant**].

In all other cases, the following rule applies

$$\frac{\begin{array}{c}\text{statEnvs} \; ; \; \text{NodeType} \; |\text{-} \; \text{child::} \; : \; \text{Type}_1 \\ \text{statEnvs} \; ; \; \text{Type}_1 \; |\text{-} \; \text{descendant::} \; : \; \text{Type}_2\end{array}}{\text{statEnvs NodeType} \; |\text{-} \; \text{descendant::} \; : \; \text{Type}_1, \; \text{Type}_2}$$

$$\frac{\begin{array}{c}\text{statEnvs} \; ; \; \text{NodeType} \; |\text{-} \; \text{self::} \; : \; \text{Type}_1 \\ \text{statEnvs} \; ; \; \text{Type}_1 \; |\text{-} \; \text{descendant::} \; : \; \text{Type}_2\end{array}}{\text{statEnvs} \; ; \; \text{NodeType} \; |\text{-} \; \text{descendant-or-self::} \; : \; \text{Type}_1, \; \text{Type}_2}$$

In all the other cases, the filter application results in an empty type.

statEnvs ; NodeType |- AxisName:: : () otherwise.

Dynamic Evaluation

The following rules define the dynamic semantics of the filter judgment when applied to an axis.

The self axis just returns the context node.

$$\frac{}{\text{dynEnvs.varValue} \text{ ; NodeValue} \vdash \text{self::} \Rightarrow \text{NodeValue}}$$

The child, parent, attribute and namespace axis are implemented through their corresponding accessors in the [XQuery 1.0 and XPath 2.0 Data Model].

$$\frac{}{\text{dynEnvs.varValue} \text{ ; NodeValue} \vdash \text{child::} \Rightarrow \text{dm:children(NodeValue)}}$$

$$\frac{}{\text{dynEnvs.varValue} \text{ ; NodeValue} \vdash \text{attribute::} \Rightarrow \text{dm:attributes(NodeValue)}}$$

$$\frac{}{\text{dynEnvs.varValue} \text{ ; NodeValue} \vdash \text{parent::} \Rightarrow \text{dm:parent(NodeValue)}}$$

The descendant, descendant-or-self, ancestor, and ancestor-or-self axis are implemented through recursive application of the children and parent filters.

$$\frac{\begin{array}{c}\text{dynEnvs.varValue} \text{ ; NodeValue} \vdash \text{child::} \Rightarrow \text{Value}_1 \\ \text{dynEnvs.varValue} \text{ ; Value}_1 \vdash \text{descendant::} \Rightarrow \text{Value}_2\end{array}}{\text{dynEnvs.varValue} \text{ ; NodeValue} \vdash \text{descendant::} \Rightarrow \text{Value}_1, \text{Value}_2}$$

$$\frac{\begin{array}{c}\text{dynEnvs.varValue} \text{ ; NodeValue} \vdash \text{self::} \Rightarrow \text{Value}_1 \\ \text{dynEnvs.varValue} \text{ ; Value}_1 \vdash \text{descendant::} \Rightarrow \text{Value}_2\end{array}}{\text{dynEnvs.varValue} \text{ ; NodeValue} \vdash \text{descendant-or-self::} \Rightarrow \text{Value}_1, \text{Value}_2}$$

$$\frac{\begin{array}{c}\text{dynEnvs.varValue} \text{ ; NodeValue} \vdash \text{parent::} \Rightarrow \text{Value}_1 \\ \text{dynEnvs.varValue} \text{ ; Value}_1 \vdash \text{ancestor::} \Rightarrow \text{Value}_2\end{array}}{\text{dynEnvs.varValue} \text{ ; NodeValue} \vdash \text{ancestor::} \Rightarrow \text{Value}_1, \text{Value}_2}$$

$$\frac{\begin{array}{c}\text{dynEnvs.varValue} \text{ ; NodeValue} \vdash \text{self::} \Rightarrow \text{Value}_1 \\ \text{dynEnvs.varValue} \text{ ; Value}_1 \vdash \text{ancestor::} \Rightarrow \text{Value}_2\end{array}}{\text{dynEnvs.varValue} \text{ ; NodeValue} \vdash \text{ancestor-or-self::} \Rightarrow \text{Value}_1, \text{Value}_2}$$

In all the other cases, the filter application results in an empty sequence.

dynEnvs.varValue ; NodeValue |- AxisName:: => () otherwise.

> **Ed. Note:** Peter: Generally steps may operate on nodes and values alike; the axis rules only can operate on nodes (NodeValue). Is it a dynamic error to apply an axis rule on a value? See [**Issue-0125: Operations on node only in XPath**].

4.3.1.2 Node Tests

Introduction

A node test is a condition applied on the nodes selected by an axis step. Possible node tests are described by the following grammar productions.

[36]	NodeTest	::=	NameTest | KindTest
[37]	NameTest	::=	QName | Wildcard
[38]	Wildcard	::=	"*" | ":"? NCName ":" "*" | "*" ":" NCName
[39]	KindTest	::=	ProcessingInstructionTest | CommentTest | TextTest | AnyKindTest
[40]	ProcessingInstructionTest	::=	"processing-instruction" "(" StringLiteral? ")"
[41]	CommentTest	::=	"comment" "(" ")"
[42]	TextTest	::=	"text" "(" ")"
[43]	AnyKindTest	::=	"node" "(" ")"

The inference rules for the filter node tests indicate, for each node step and each input node, whether the node should be kept or left out. Therefore, each rule either returns the input node, or returns the empty sequence. The overall result is then obtained by putting together all of the remaining nodes, following the generic semantics for filters.

Static Type Analysis

Node tests on elements and attributes always accomplish the most specific type possible. For example, if `$v` is bound to an element with a computed name, the type of `$v` is `element * { Type }`. The static type computed for the expression `$v/self::foo` is `element foo {Type}`, which makes use of the nametest `foo` to arrive at a more specific type.

$$\frac{qname_2 = \text{prefix}_2\text{:local}_2 \quad \text{xf:get-namespace-uri}(\ qname_1) = \text{statEnvs.varType}(\text{prefix}_2) \quad \text{xf:get-local-name}(\ qname_1\ ) = \text{local}_2}{\text{statEnvs}\ ;\ \text{element}\ qname_1\ \{\text{Type}\} \vdash qname_2 : \text{element}\ qname_1\ \{\text{Type}\}}$$

$$\frac{qname_2 = \text{prefix}_2\text{:local}_2 \quad \text{local}_1 = \text{local}_2}{\text{statEnvs}\ ;\ \text{element}\ *\text{:local}_1\ \{\text{Type}\} \vdash qname_2 : \text{element}\ qname_2\ \{\text{Type}\}}$$

$$\frac{qname_2 = \text{prefix}_2\text{:local}_2 \quad \text{statEnvs.namespace}(\text{prefix}_1) = \text{statEnvs.namespace}(\text{prefix}_2)}{\text{statEnvs}\ ;\ \text{element}\ \text{prefix}_1\text{:*}\ \{\text{Type}\} \vdash qname_2 : \text{element}\ \text{prefix}_1\text{:local}_2\{\text{Type}\}}$$

$$\frac{}{\text{statEnvs}\ ;\ \text{element}\ *\ \{\text{Type}\} \vdash qname_2 : \text{element}\ qname_2\ \{\text{Type}\}}$$

$$\frac{\text{xf:get-local-name(}\ qname_1\ \text{)} = local_2}{\text{statEnvs ; element } qname_1 \text{ \{Type\} |- *:}local_2 \text{ : element } qname_1 \text{ \{Type\}}}$$

$$\frac{local_1 = local_2}{\text{statEnvs ; element *:}local_1 \text{ \{Type\} |- *:}local_2 \text{ : element *:}local_2 \text{ \{Type\}}}$$

$$\frac{}{\text{statEnvs ; element prefix}_1\text{:* \{Type\} |- *:}local_2\text{: element prefix}_1\text{:}local_2\text{\{Type\}}}$$

$$\frac{}{\text{statEnvs ; element * \{ Type \} |- *:}local_2 \text{ : element *:}local_2 \text{ \{Type\}}}$$

$$\frac{\text{xf:get-namespace-uri(}\ qname_1\text{)} = \text{statEnvs.namespace(prefix}_2\text{)}}{\text{statEnvs ; element } qname_1 \text{ \{Type\} |- prefix}_2\text{:* : element } qname_1 \text{ \{Type\}}}$$

$$\frac{}{\text{statEnvs ; element *:}local_1 \text{ \{Type\} |- prefix}_2\text{:* : element prefix}_2\text{:}local_1 \text{ \{Type\}}}$$

$$\frac{\text{statEnvs.namespace(prefix}_1\text{)} = \text{statEnvs.namespace(prefix}_2\text{)}}{\text{statEnvs ; element prefix}_1\text{:* \{Type\} |- prefix}_2\text{:* : element prefix}_1\text{:* \{Type\}}}$$

$$\frac{}{\text{statEnvs ; element * \{Type\} |- prefix}_2\text{:* : element prefix}_2\text{:* \{Type\}}}$$

$$\frac{}{\text{statEnvs ; element prefix:local \{Type\} |- * : element prefix:local \{Type\}}}$$

Very similar typing rules apply to attributes:

$$\frac{\begin{array}{c} qname_2 = \text{prefix}_2\text{:}local_2 \\ \text{xf:get-namespace-uri(}\ qname_1\text{)} = \text{statEnvs.namespace(prefix}_2\text{)} \\ \text{xf:get-local-name(}\ qname_1\ \text{)} = local_2 \end{array}}{\text{statEnvs ; attribute } qname_1 \text{ \{Type\} |- } qname_2 \text{ : attribute } qname_1 \text{ \{Type\}}}$$

$$\frac{\begin{array}{c} qname_2 = \text{prefix}_2\text{:}local_2 \\ local_1 = local_2 \end{array}}{\text{statEnvs ; attribute *:}local_1 \text{ \{Type\} |- } qname_2 \text{ : attribute } qname_2 \text{ \{Type\}}}$$

$$\frac{\begin{array}{c} qname_2 = \text{prefix}_2\text{:}local_2 \\ \text{statEnvs.namespace(prefix}_1\text{)} = \text{statEnvs.namespace(prefix}_2\text{)} \end{array}}{}$$

statEnvs ; attribute prefix$_1$:* {Type} |- *qname*$_2$: attribute prefix$_1$:local$_2${Type}

---

statEnvs ; attribute * {Type} |- *qname*$_2$: attribute *qname*$_2$ {Type}

xf:get-local-name(*qname*$_1$) = local$_2$

---

statEnvs ; attribute *qname*$_1$ {Type} |- *:local$_2$: attribute *qname*$_1$ {Type}

local$_1$ = local$_2$

---

statEnvs ; attribute *:local$_1$ {Type} |- *:local$_2$: attribute *:local$_2$ {Type}

---

statEnvs ; attribute prefix$_1$:* {Type} |- *:local$_2$: attribute prefix$_1$:local$_2${Type}

---

statEnvs ; attribute * {Type} |- *:local$_2$: attribute *:local$_2$ {Type}

xf:get-namespace-uri(*qname*$_1$) = statEnvs.namespace(prefix$_2$)

---

statEnvs ; attribute *qname*$_1$ {Type} |- prefix$_2$:* : attribute *qname*$_1$ {Type}

---

statEnvs ; attribute *:local$_1$ {Type} |- prefix$_2$:* : attribute prefix$_2$:local$_1$ {Type}

statEnvs.namespace(prefix$_1$) = statEnvs.namespace(prefix$_2$)

---

statEnvs ; attribute prefix$_1$:* {Type} |- prefix$_2$:* : attribute prefix$_1$:* {Type}

---

statEnvs ; attribute * {Type} |- prefix$_2$:* : attribute prefix$_2$:* {Type}

---

statEnvs ; attribute prefix:local {Type} |- * : attribute prefix:local {Type}

Comments, processing instructions, and text:

---

statEnvs ; processing-instruction |- processing-instruction () : processing-instruction

---

statEnvs ; processing-instruction |- processing-instruction (string) : processing-instruction

---

statEnvs ; comment |- comment () : comment

$$\frac{}{\text{statEnvs} \;;\; \text{text} \;|\!-\; \text{text}\ () : \text{text}}$$

$$\frac{}{\text{statEnvs} \;;\; \text{NodeType} \;|\!-\; \text{node}\ () : \text{NodeType}}$$

If none of the above rules applies then the node test returns the empty sequence, and the following dynamic rule is applied:

$$\frac{}{\text{statEnvs} \;;\; \text{NodeType} \;|\!-\; \text{node}\ () : ()}$$

Ed. Note: Peter: Except for `self::`, all axes guarantee that the `NodeType` is not the generic type`node`. However, when the type `node` is encountered, it has to be interpreted as `element | attribute | text | comment | processing-instruction` for these typing rules to work.

Dynamic Evaluation

$$\frac{\begin{array}{c}\text{dm:name}(\ \text{NodeValue}\ ) = \mathit{qname}_1 \\ \mathit{qname}_2 = \text{prefix}_2\text{:local}_2 \\ \text{xf:get-namespace-uri}\ (\ \mathit{qname}_1\ ) = \text{statEnvs.namespace}(\text{prefix}_2) \\ \text{xf:get-local-name}\ (\ \mathit{qname}_1\ ) = \text{local}_2\end{array}}{\text{dynEnvs.varValue} \;;\; \text{NodeValue} \;|\!-\; \mathit{qname}_2 \Rightarrow \text{NodeValue}}$$

$$\frac{\text{dm:name}\ (\ \text{NodeValue}\ ) = \mathit{qname}}{\text{dynEnvs.varValue} \;;\; \text{NodeValue} \;|\!-\; * \Rightarrow \text{NodeValue}}$$

$$\frac{\begin{array}{c}\text{dm:name}\ (\ \text{NodeValue}\ ) = \mathit{qname} \\ \text{xf:get-namespace-uri}\ (\ \mathit{qname}\ ) = \text{statEnvs.namespace}(\text{prefix})\end{array}}{\text{dynEnvs.varValue} \;;\; \text{NodeValue} \;|\!-\; \text{prefix:*} \Rightarrow \text{NodeValue}}$$

$$\frac{\begin{array}{c}\text{dm:name}\ (\ \text{NodeValue}\ ) = \mathit{qname} \\ \text{xf:get-local-name}\ (\ \mathit{qname}\ ) = \text{local}\end{array}}{\text{dynEnvs.varValue} \;;\; \text{NodeValue} \;|\!-\; \text{*:local} \Rightarrow \text{NodeValue}}$$

$$\frac{\text{dm:node-kind}\ (\ \text{NodeValue}\ ) = \text{"processing-instruction"}}{\text{dynEnvs.varValue} \;;\; \text{NodeValue} \;|\!-\; \text{processing-instruction}\ () \Rightarrow \text{NodeValue}}$$

$$\frac{\begin{array}{c}\text{dm:node-kind}\ (\ \text{NodeValue}\ ) = \text{"processing-instruction"} \\ \text{dm:name}\ (\ \text{NodeValue}\ ) = \mathit{qname} \\ \text{xf:get-local-name}\ (\ \mathit{qname}\ ) = \text{String}\end{array}}{\text{dynEnvs.varValue} \;;\; \text{NodeValue} \;|\!-\; \text{processing-instruction}\ (\ \text{String}\ ) \Rightarrow \text{NodeValue}}$$

Ed. Note: Note the use of the `xf:get-local-name` function to extract the local name out of

the name of the node.

$$\frac{\text{dm:node-kind (NodeValue) = "comment"}}{\text{dynEnvs.varValue ; NodeValue |- comment () => NodeValue}}$$

$$\frac{\text{dm:node-kind (NodeValue) = "text"}}{\text{dynEnvs.varValue ; NodeValue |- text () => NodeValue}}$$

The `node()` node test is true for all nodes. Therefore, the following rule does not have any precondition (remember that an empty upper part in the rule indicates that the rule is always true).

$$\frac{}{\text{dynEnvs.varValue ; NodeValue |- node () => NodeValue}}$$

If none of the above rules applies then the node test returns the empty sequence, and the following dynamic rule is applied:

$$\text{dynEnvs.varValue ; NodeValue |- node () => ()}$$

4.3.2 General Steps

Introduction

General steps are composed of primary expressions, followed by a step qualifier. The semantics of step qualifiers is defined in the next subsection.

[44] GeneralStep ::= PrimaryExpr StepQualifiers

Normalization

Primary expressions are normalized as expressions. I.e., their Path normalization is obtained by applying the general normalization rules for expressions. In GeneralSteps, StepQualifiers are always applied as forward steps. Note that the following rule makes use of a typewitch expression to ensure the path expression is always applied on a sequence of nodes.

> **Ed. Note:** Jerome: where to enforce the restriction that XPath expressions operate on nodes is still an open issue. See **[Issue-0125: Operations on node only in XPath]**.

$$\left[\text{PrimaryExpr}\right]_{\text{Path}} == \text{typeswitch } \left[\text{PrimaryExpr}\right]_{\text{Expr}} \text{ as \$fs:new}$$
case node* return $fs:new
default return dm:error()

$$[\text{PrimaryExpr StepQualifiers}]_{\text{Path}}$$

$$==$$

$$[[\text{PrimaryExpr}]_{\text{Path}} \text{ StepQualifiers}]_{\text{ForwardPath}}$$

4.3.3 Step Qualifiers

Introduction

Step qualifiers are composed of zero or more predicates (using the `[ ... ]` notation) or dereference operations (using the `=>` notation).

[46] StepQualifiers ::= (("[" Expr "]") | ("=>" NameTest))*

Normalization

Normalization is first applied on all the components of a step qualifier. Remember that StepQualifiers might be normalized through either a ForwardPath rule, or a ReversePath rule.

$$[\text{StepQualifier}_1 \text{ StepQualifier}_2 \ldots]_{\text{ForwardPath}}$$
$$==$$
$$[\text{StepQualifier}_1]_{\text{ForwardPath}} [\text{StepQualifier}_2]_{\text{ForwardPath}} \cdots$$

$$[\text{StepQualifier}_1 \text{ StepQualifier}_2 \ldots]_{\text{ReversePath}}$$
$$==$$
$$[\text{StepQualifier}_1]_{\text{ReversePath}} [\text{StepQualifier}_2]_{\text{ReversePath}} \cdots$$

4.3.3.1 Predicates

Normalization

We now define the semantics of predicates, for both forward and reverse steps.

$$[\text{Expr}_1 [\text{Expr}_2]]_{\text{ForwardPath}}$$
$$==$$

let $fs:sequence := $[\text{Expr}_1]_{\text{Path}}$ return
let $fs:last := xf:length(Sequence) return
for $fs:position in op:to(1,$fs:last) return
let $fs:last := xf:length($fs:sequence) return
let $fs:dot := fs:unique-item-at($fs:sequence, $fs:position) return
if $[\text{Expr}_2]_{\text{Predicate}}$ then $fs:dot else ()

$$[\text{Expr}_1\ [\ \text{Expr}_2\ ]\ ]_{\text{ReversePath}}$$
==

let $fs:sequence := $[\text{Expr}_1]_{\text{Path}}$ return
let $fs:last := xf:length(Sequence)
for $fs:counter in op:to(1,$fs:last) return
let $fs:position := $fs:last + 1 - $fs:counter return
let $fs:last := 1 return
let $fs:dot := fs:unique-item-at($fs:sequence, $fs:position)
return $[\text{Expr}_2]_{\text{Predicate}}$) then $fs:dot else ()

Predicates in path expressions are normalized with a special mapping rule:

$$[\text{Expr}]_{\text{Predicate}}$$
==

typeswitch $[\text{Expr}]$ as $v
case () return xf:false()
case numeric return op:numeric-equal(xf:round($v), $fs:position)
case xs:boolean return $v
default return
if (xf:length($[\text{\$v/self::node()}]_{\text{Path}}$)>=1)
then xf:true()
else dm:error()

4.3.3.2 Dereferences

Normalization

Dereference steps are mapped into an iteration, followed by a call to the dereference function, then followed by the appropriate NameTest

$$[\ \text{ForwardStep "=>" NameTest}\ ]_{\text{Path}}$$

==

for $fs:dot in [ForwardStep]_Expr return
let $fs:dot := dm:dereference($fs:dot) return
self::NameTest

4.3.4 Unabbreviated Syntax

Ed. Note: XQuery Section 2.3.4 has no semantic content -- it just contains examples! (This suggests that perhaps the section structure is a little weird.)

4.3.5 Abbreviated Syntax

[45] AbbreviatedStep ::= "." | ".." | ("@" NameTest StepQualifiers)

Normalization

Here are normalization rules for the abbreviated syntax.

$$[\ \text{// RelativePathExpr}\ ]_{\text{Path}}$$

==

$$\text{/ descendant-or-self::node() / } [\text{RelativePathExpr}]_{\text{Path}}$$

$$[\ \text{StepExpr // RelativePathExpr}\ ]_{\text{Path}}$$

==

$$[\text{StepExpr}]_{\text{Step}} \text{ / descendant-or-self::node() / } [\text{RelativePathExpr}]_{\text{Path}}$$

$$[\ .\ ]_{\text{Path}}$$

==

$$[\ ..\ ]_{\text{Path}}$$

==

parent::node()

$$[\ \text{@ NodeTest}\ ]_{\text{Path}}$$

==

attribute :: NodeTest

$$[\ \text{NodeTest}\ ]_{\text{Path}}$$

==

child :: NodeTest

4.4 Sequence Expressions

Introduction

This section defines the semantics of [2.4 Sequence Expressions] in [XQuery 1.0: A Query Language for XML].

XQuery supports operators to construct and combine sequences. A **sequence** is an ordered collection of zero or more items. An **item** is either an atomic value or a node.

4.4.1 Constructing Sequences

[50]	ParenthesizedExpr	::=	"(" ExprSequence? ")"
[3]	ExprSequence	::=	Expr ("," Expr)*
[17]	RangeExpr	::=	Expr "to" Expr

Normalization

The sequence expression is normalized into a sequence of core expressions:

$$[\ \text{"("}\ \textit{ExprSequence}\ \text{")"}\ ]$$

==

$$\text{"("}\ [\ \textit{ExprSequence}\ ]\ \text{")"}$$

$$[\ \textit{Expr}_1\ ,\ \ldots\ ,\ \textit{Expr}_n\ ]$$

==

$$[\ \textit{Expr}_1\ ],\ \ldots,\ [\ \textit{Expr}_n\ ]$$

> **Ed. Note:** Mike Kay remarks that it would be cleaner to have binary rules and recursion to define the semantics rather than the ... notation.

Core Grammar

The core grammar rule for sequence expressions is:

[3]	ExprSequence	::=	Expr ("," Expr)*

Static Type Analysis

The static semantics of the sequence expression follows. The type of the sequence expression is the sequence over the types of individual expressions.

$$\frac{\text{statEnvs} \mid\!- Expr_1 : Type_1 \quad \ldots \quad \text{statEnvs} \mid\!- Expr_n : Type_n}{\text{statEnvs} \mid\!- Expr_1 , \ldots , Expr_n : Type_1, \ldots, Type_n}$$

Dynamic Evaluation

The dynamic semantics of the sequence expression follows. Each expression in the sequence is evaluated and the resulting values are concatenated into one sequence.

$$\frac{\text{dynEnvs} \mid\!- Expr_1 \Rightarrow value_1 \quad \text{dynEnvs} \mid\!- Expr_2 \Rightarrow value_2 \quad \ldots \quad \text{dynEnvs} \mid\!- Expr_n \Rightarrow value_n}{\text{dynEnvs} \mid\!- Expr_1, \ldots, Expr_n \Rightarrow \text{op:concatenate}(value_1, \text{op:concatenate}(value_2, \text{op:concatenate}(\ldots , value_n)\ldots)}$$

Normalization

The normalization of the infix "to" operator maps it into an application of op:to.

$$\left[Expr_1 \text{ "to" } Expr_2 \right]$$
$$==$$
$$\left[\text{op:to } (Expr_1, Expr_2) \right]$$

Static Type Analysis

The static semantics of the op:to function is defined in **[6 Additional Semantics of Functions]**.

Dynamic Evaluation

The dynamic semantics rules for function calls given in **[4.2.4 Function Calls]** are applied to the function call op:to above.

> **Ed. Note:** Should the "to" operator be defined formally? See [**Issue-0133: Should to also be described in the formal semantics?**].

4.4.2 Combining Sequences

XQuery provides several operators for combining sequences.

[20]	UnionExpr	::=	Expr ("union" | "|") Expr
[21]	IntersectExceptExpr	::=	Expr ("intersect" | "except") Expr

Notation

First, a union, intersect, or except expression is normalized into a corresponding core expression. The functions $[\,]_{\text{SequenceOp}}$ and $[\,]_{\text{SequenceValueOp}}$ are defined by the following tables:

SequenceOp	$[\text{SequenceOp}]_{\text{SequenceOp}}$
"union"	op:union
"|"	op:union
"intersect"	op:intersect
"except"	op:except

SequenceOp	$[\text{SequenceOp}]_{\text{SequenceValueOp}}$
"union"	op:union-value
"|"	op:union-value
"intersect"	op:intersect-value
"except"	op:except-value

Normalization

$$[\,\textit{Expr}_1 \text{ SequenceOp } \textit{Expr}_2\,] \quad == \quad$$

```
typeswitch (Expr1) as $v1
case fs:atomic+ return
   (typeswitch (Expr2) as $v2
   case fs:atomic+ return [SequenceOp]SequenceValueOp($v1, $v2)
   default return dm:error())
case node+
   (typeswitch (Expr2) as $v2
   case node+ return [SequenceOp]SequenceOp($v1, $v2)
   default return dm:error())
default return dm:error()
```

Ed. Note: MFF: this mapping assumes that there are two versions of union: one based on node identity and one based on value equality. What happens if we have a heterogeneous sequence is not clear. See **[Issue-0120: Sequence operations: value vs. node identity]**.

Static Type Analysis

The static semantics of the functions that operate on sequences are defined in **[6 Additional Semantics of Functions]**.

Dynamic Evaluation

The dynamic semantics rules for function calls given in **[4.2.4 Function Calls]** are applied to the calls to

functions on sequences above.

4.5 Arithmetic Expressions

This section defines the semantics of [2.5 Arithmetic Expressions] in [XQuery 1.0: A Query Language for XML].

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

[18]	AdditiveExpr	::=	Expr ("+" | "-") Expr
[19]	MultiplicativeExpr	::=	Expr ("*" | "div" | "mod") Expr
[22]	UnaryExpr	::=	("-" | "+") Expr

Notation

> **Ed. Note:** MFF: The operator table in this section was produced by Don Chamberlin. This table should be in one place, probably the formal semantics

The tables in this section list the combinations of datatypes for which the various operators of XQuery are defined. For each valid combination of datatypes, the table indicates the name of the function that implements the operator and the datatype of the result. Definitions of the functions can be found in [XQuery 1.0 and XPath 2.0 Functions and Operators].

In the following tables, the term fs:numeric refers to the types xs:integer, xs:decimal, xs:float, and xs:double. When the result type of an operator is listed as fs:numeric, it means "same as the highest type of any input operand, in promotion order." For example, when invoked with operands of type xs:integer and xs:float, the binary + operator returns a result of type xs:float.

In the following tables, the term Gregorian refers to the types xs:gYearMonth, xs:gYear, xs:gMonthDay, xs:gDay, and xs:gMonth. For binary operators that accept two Gregorian-type operands, both operands must have the same type (for example, if one operand is of type xs:gDay, the other operand must be of type xs:gDay.)

The functions $[]_{\text{BinaryOp}}$ and $[]_{\text{UnaryOp}}$ are defined by the following two tables. The function $[]_{\text{BinaryOp}}$ takes the left-hand expression (A) and its type, the operator, the right-hand expression (b) and its type and returns a new expression, which applies the type-appropriate operator to the two expressions. The function $[]_{\text{UnaryOp}}$ takes an operator and an expression and returns a new expression, which applies the appropriate operator to the expression.

Operator	**Type(A)**	**Type(B)**	**$[$A; Type(A); Operator; B; Type(B)$]_{\text{BinaryOp}}$**	**Result type**
A + B	fs:numeric	fs:numeric	op:numeric-add(A, B)	fs:numeric
A + B	xs:date	xs:duration	(missing)	xs:date
A + B	xs:duration	xs:date	(missing)	xs:date

A + B	xs:time	xs:duration	(missing)	xs:time
A + B	xs:duration	xs:time	(missing)	xs:time
A + B	xs:dateTime	xs:duration	op:get-end-datetime(A, B)	xs:dateTime
A + B	xs:duration	xs:dateTime	op:get-end-datetime(B, A)	xs:dateTime
A + B	xs:duration	xs:duration	(missing)	xs:duration
A - B	fs:numeric	fs:numeric	op:numeric-subtract(A, B)	fs:numeric
A - B	xs:date	xs:date	(missing)	xs:duration
A - B	xs:date	xs:duration	(missing)	xs:date
A - B	xs:time	xs:time	(missing)	xs:duration
A - B	xs:time	xs:duration	(missing)	xs:time
A - B	xs:dateTime	xs:dateTime	op:get-duration(A, B)	xs:duration
A - B	xs:dateTime	xs:duration	op:get-start-datetime(A, B)	xs:dateTime
A - B	xs:duration	xs:duration	(missing)	xs:duration
A * B	fs:numeric	fs:numeric	op:numeric-multiply(A, B)	fs:numeric
A div B	fs:numeric	fs:numeric	op:numeric-divide(A, B)	fs:numeric
A mod B	fs:numeric	fs:numeric	op:numeric-mod(A, B)	fs:numeric
A eq B	fs:numeric	fs:numeric	op:numeric-equal(A, B)	xs:boolean
A eq B	xs:boolean	xs:boolean	op:boolean-equal(A, B)	xs:boolean
A eq B	xs:string	xs:string	op:numeric-equal(xf:compare(A, B), 1)	xs:boolean
A eq B	xs:date	xs:date	(missing)	xs:boolean?
A eq B	xs:time	xs:time	(missing)	xs:boolean?
A eq B	xs:dateTime	xs:dateTime	op:datetime-equal(A, B)	xs:boolean?
A eq B	xs:duration	xs:duration	op:duration-equal(A, B)	xs:boolean?
A eq B	Gregorian	Gregorian	(missing)	xs:boolean
A eq B	xs:hexBinary	xs:hexBinary	op:hex-binary-equal(A, B)	xs:boolean
A eq B	xs:base64Binary	xs:base64Binary	op:base64-binary-equal(A, B)	xs:boolean
A eq B	xs:anyURI	xs:anyURI	(missing)	xs:boolean
A eq B	xs:QName	xs:QName	(missing)	xs:boolean
A ne B	fs:numeric	fs:numeric	xf:not(op:numeric-equal(A, B))	xs:boolean
A ne B	xs:boolean	xs:boolean	xf:not(op:boolean-equal(A, B))	xs:boolean
A ne B	xs:string	xs:string	xf:not(op:numeric-equal(xf:compare(A, B), 1))	xs:boolean
A ne B	xs:date	xs:date	(missing)	xs:boolean?
A ne B	xs:time	xs:time	(missing)	xs:boolean?
A ne B	xs:dateTime	xs:dateTime	xf:not3(op:datetime-equal(A, B))	xs:boolean?
A ne B	xs:duration	xs:duration	xf:not3(op:duration-equal(A, B))	xs:boolean?
A ne B	Gregorian	Gregorian	(missing)	xs:boolean
A ne B	xs:hexBinary	xs:hexBinary	xf:not(op:hex-binary-equal(A, B))	xs:boolean
A ne B	xs:base64Binary	xs:base64Binary	xf:not(op:base64-binary-equal(A, B))	xs:boolean
A ne B	xs:anyURI	xs:anyURI	(missing)	xs:boolean
A ne B	xs:QName	xs:QName	(missing)	xs:boolean
A eq B	xs:NOTATION	xs:NOTATION	(missing)	xs:boolean

A ne B	xs:NOTATION	xs:NOTATION	(missing)	xs:boolean
A gt B	fs:numeric	fs:numeric	op:numeric-greater-than(A, B)	xs:boolean
A gt B	xs:boolean	xs:boolean	(missing)	xs:boolean
A gt B	xs:string	xs:string	op:numeric-greater-than(xf:compare(A, B), 0)	xs:boolean
A gt B	xs:date	xs:date	(missing)	xs:boolean?
A gt B	xs:time	xs:time	(missing)	xs:boolean?
A gt B	xs:dateTime	xs:dateTime	op:datetime-greater-than(A, B)	xs:boolean?
A gt B	xs:duration	xs:duration	op:duration-greater-than(A, B)	xs:boolean?
A lt B	fs:numeric	fs:numeric	op:numeric-less-than(A, B)	xs:boolean
A lt B	xs:boolean	xs:boolean	(missing)	xs:boolean
A lt B	xs:string	xs:string	op:numeric-less-than(xf:compare(A, B), 0)	xs:boolean
A lt B	xs:date	xs:date	(missing)	xs:boolean?
A lt B	xs:time	xs:time	(missing)	xs:boolean?
A lt B	xs:dateTime	xs:dateTime	op:datetime-less-than(A, B)	xs:boolean?
A lt B	xs:duration	xs:duration	op:duration-less-than(A, B)	xs:boolean?
A ge B	fs:numeric	fs:numeric	op:numeric-less-than(B, A)	xs:boolean
A ge B	xs:string	xs:string	op:numeric-greater-than(xf:compare(A, B), -1)	xs:boolean
A ge B	xs:date	xs:date	(missing)	xs:boolean?
A ge B	xs:time	xs:time	(missing)	xs:boolean?
A ge B	xs:dateTime	xs:dateTime	op:datetime-less-than(B, A)	xs:boolean?
A ge B	xs:duration	xs:duration	op:duration-less-than(B, A)	xs:boolean?
A le B	fs:numeric	fs:numeric	op:numeric-greater-than(B, A)	xs:boolean
A le B	xs:string	xs:string	op:numeric-less-than(xf:compare(A, B), 1)	xs:boolean
A le B	xs:date	xs:date	(missing)	xs:boolean?
A le B	xs:time	xs:time	(missing)	xs:boolean?
A le B	xs:dateTime	xs:dateTime	op:datetime-greater-than(B, A)	xs:boolean?
A le B	xs:duration	xs:duration	op:duration-greater-than(B, A)	xs:boolean?
A == B	node	node	op:node-equal(A, B)	xs:boolean
A !== B	node	node	xf:not(op:node-equal(A, B))	xs:boolean
A << B	node	node	op:node-before(A, B)	xs:boolean
A >> B	node	node	op:node-after(A, B)	xs:boolean
A precedes B	node	node	(missing)	xs:boolean
A follows B	node	node	(missing)	xs:boolean
A union B	item*	item*	op:union(A, B)	item*
A | B	item*	item*	op:union(A, B)	item*
A intersect B	item*	item*	op:intersect(A, B)	item*
A except B	item*	item*	op:except(A, B)	item*
A to B	xs:decimal	xs:decimal	op:to(A, B)	xs:integer+

A , B	item*	item*	op:concatenate(A, B)	item*

An analogous table exists for unary operators.

Operator	Operand type	$[\text{Operator; Expr}]_{\text{UnaryOp}}$	Result type
+ A	fs:numeric	op:numeric-unary-plus(A)	same as operand
- A	fs:numeric	op:numeric-unary-minus(A)	same as operand

Normalization

The normalization rules for the arithmetic operators "+" and "-" are similar, but not identical, because as the table above illustrates, "-" is not commutative.

The following normalization rule for "+" first applies $[]_{\text{Optional\_Atomic\_Type}}$ to each argument expression, binding the results of these expressions to two new variables, `$e1` and `$e2`. It then applies a typeswitch on the left-hand operand `$e1`, and for each left-hand operand type, it applies a second typeswitch on the right-hand operand `$e2`. The function $[\text{Operator}]_{\text{BinaryOp}}$ takes the operator, the left-hand type, and the right-hand type and returns the appropriate function, which is applied to the argument values.

$$[\![Expr_1 \text{ "+" } Expr_2]\!] \quad ==$$

```
let $coree1 := ([ Expr1 ]),
    $coree2 := ([ Expr2 ]),
    $e1 := [ $coree1 ]Optional_Atomic_Value,
    $e2 := [ $coree2 ]Optional_Atomic_Value
return
typeswitch ($e1) as $v1
case () return ()
case fs:numeric return
    (typeswitch ($e2) as $v2
    case () return ()
        case fs:numeric return [ $v1; fs:numeric; "+"; $v2; fs:numeric ]BinaryOp
    case fs:UnknownSimpleType return
```

$[\$v1; fs:numeric; "+"; (cast (\$v2) as xs:double); fs:numeric]_{BinaryOp}$

default return dm:error())
case xs:date return
(typeswitch ($e2) as $v2
case xs:duration return $[\$v1; xs:date; "+"; \$v2; xs:duration]_{BinaryOp}$
case fs:UnknownSimpleType return
$[\$v1; xs:date; "+"; (cast (\$v2) as xs:duration); xs:duration]_{BinaryOp}$
default return dm:error())
case xs:time return
(typeswitch ($e2) as $v2
case xs:duration return $[\$v1; xs:time; "+"; \$v2; xs:duration]_{BinaryOp}$
case fs:UnknownSimpleType return
$[\$v1; xs:time; "+"; (cast (\$v2) as xs:duration); xs:duration]_{BinaryOp}$
default return dm:error())
case xs:dateTime return
(typeswitch ($e2) as $v2
case xs:duration return $[\$v1; xs:dateTime; "+"; \$v2; xs:duration]_{BinaryOp}$
case fs:UnknownSimpleType return
$[\$v1; xs:dateTime; "+"; (cast (\$v2) as xs:duration); xs:duration]_{BinaryOp}$
default return dm:error())
case xs:duration return
(typeswitch ($e2) as $v2
case xs:duration return $[\$v1; xs:duration; "+"; \$v2; xs:duration]_{BinaryOp}$
case xs:date return $[\$v1; xs:duration; "+"; \$v2; xs:date]_{BinaryOp}$
case xs:time return $[\$v1; xs:duration; "+"; \$v2; xs:time]_{BinaryOp}$
case xs:dateTime return $[\$v1; xs:duration; "+"; \$v2; xs:dateTime]_{BinaryOp}$
case fs:UnknownSimpleType return

[$v1; xs:duration; "+"; (cast ($v2) as xs:duration); xs:duration]$_{\text{BinaryOp}}$
default return dm:error())
case fs:UnknownSimpleType return
(typeswitch ($e2) as $v2
case () return ()
case fs:numeric return
[(cast ($v1) as xs:double); xs:double; "+"; $v2; xs:double]$_{\text{BinaryOp}}$
case xs:duration return

[(cast ($v1) as xs:duration); xs:duration; "+"; $v2; xs:duration;]$_{\text{BinaryOp}}$

case xs:time
[(cast ($v1) as xs:duration); xs:duration; "+"; $v2; xs:time]$_{\text{BinaryOp}}$
case xs:dateTime return
[(cast ($v1) as xs:duration); xs:duration; "+"; $v2; xs:dateTime]$_{\text{BinaryOp}}$
case fs:UnknownSimpleType return
[(cast ($v1) as xs:double); xs:double; "+";(cast ($v2) as xs:double); xs:double]$_{\text{BinaryOp}}$
default return dm:error())
default return dm:error()

Ed. Note: MFF: The XQuery document specifies that the casting should depend on the lexical form. This cannot be captured by normalization. See **[Issue-0128: Casting based on the lexical form]**.

Ed. Note: MFF: The Datatype production does not permit choices of item types -- this is annoying. See **[Issue-0127: Datatype limitations]**.

Ed. Note: Peter: the static semantics of operators is as strict as the one of functions, and is still under discussion. See **[Issue-0129: Static typing of union]**.

The following normalization rule for "-" is analogous to that for "+".

$$[\,Expr_1\ \text{"-"}\ Expr_2\,]$$
==

let $coree1 := ([$Expr_1$]),
$coree2 := ([$Expr_2$]),
$e1 := [$coree1]$_{Optional_Atomic_Value}$,
$e2 := [$coree2]$_{Optional_Atomic_Value}$
return
typeswitch ($e1) as $v1
case () return ()
case fs:numeric return
(typeswitch ($e2) as $v2
case () return ()
case fs:numeric return [$v1; fs:numeric; "-"; $v2; fs:numeric]$_{BinaryOp}$
case fs:UnknownSimpleType return
[$v1; fs:numeric; "-"; (cast ($v2) as xs:double); fs:numeric]$_{BinaryOp}$
default return dm:error())
case xs:date return
(typeswitch ($e2) as $v2
case xs:duration return [$v1; xs:date; "-"; $v2; xs:duration]$_{BinaryOp}$
case fs:UnknownSimpleType return
[$v1; xs:date; "-"; (cast ($v2) as xs:duration); xs:duration]$_{BinaryOp}$
default return dm:error())
case xs:time return
(typeswitch ($e2) as $v2
case xs:duration return [$v1; xs:time; "-"; $v2; xs:duration]$_{BinaryOp}$
case fs:UnknownSimpleType return
[$v1; xs:time; "-"; (cast ($v2) as xs:duration); xs:duration]$_{BinaryOp}$
default return dm:error())
case xs:dateTime return
(typeswitch ($e2) as $v2
case xs:duration return [$v1; xs:dateTime; "-"; $v2; xs:duration]$_{BinaryOp}$
case fs:UnknownSimpleType return

$[\$v1; xs:dateTime; "-"; (cast (\$v2) as xs:duration);xs:duration]_{BinaryOp}$

default return dm:error())
case xs:duration return
(typeswitch (\$e2) as \$v2
case xs:duration return $[\$v1; xs:duration; "-"; \$v2; xs:duration]_{BinaryOp}$
case fs:UnknownSimpleType return
$[\$v1; xs:duration; "-"; (cast (\$v2) as xs:duration); xs:duration]_{BinaryOp}$
default return dm:error())
case fs:UnknownSimpleType return
(typeswitch (\$e2) as \$v2
case () return ()
case fs:numeric return
$[(cast (\$v1) as xs:double); xs:double; "-" \$v2; fs:numeric]_{BinaryOp}$
case xs:duration return
$[(cast (\$v1) as xs:duration); xs:duration; "-"; \$v2; xs:duration]_{BinaryOp}$
case xs:time return
$[(cast (\$v1) as xs:duration); xs:duration; "-"; \$v2 ; xs:time]_{BinaryOp}$
case xs:dateTime return
$[(cast (\$v1) as xs:duration); xs:duration; "-"; \$v2; xs:dateTime]_{BinaryOp}$
case fs:UnknownSimpleType return
$[(cast (\$v1) as xs:double); xs:double; "-"; (cast (\$v2) as xs:double)' xs:double]_{BinaryOp}$
default return dm:error())
default return dm:error()

The multiplicative operators "*", "div", and "mod" are only defined on fs:numeric, so their normalization rule is simple. For convenience, MultOp denotes "*", "div", or "mod".

$$[Expr_1 \text{ MultOp } Expr_2]$$
==

let $coree1 := ([$Expr_1$]),
$coree2 := ([$Expr_2$]),
$e1 := [$coree1]$_{Optional_Atomic_Value}$,
$e2 := [$coree2]$_{Optional_Atomic_Value}$
return
typeswitch ($e1) as $v1
case () return ()
case fs:numeric return
(typeswitch ($e2) as $v2
case () return ()
case fs:numeric return [$v1; fs:numeric; MultOp; $v2; fs:numeric]$_{BinaryOp}$
case fs:UnknownSimpleType return
[$v1; fs:numeric; MultOp; (cast ($v2) as xs:double); fs:numeric]$_{BinaryOp}$
default return dm:error())
case fs:UnknownSimpleType return
(typeswitch ($e2) as $v2
case () return ()
case fs:numeric return
[(cast ($v1) as xs:double); xs:doube; "-"; $v2; fs:numeric]$_{BinaryOp}$
case fs:UnknownSimpleType return
[(cast ($v1) as xs:double); xs:double; "-"; (cast ($v2) as xs:double); xs:double]$_{BinaryOp}$
default return dm:error())
default return dm:error()

Ed. Note: MFF: The XQuery document specifies that the casting should depend on the lexical form. This cannot be captured by normalization. See [**Issue-0128: Casting based on the lexical form**].

For convenience, UnaryOp denotes the unary operators "+" and "-". The normalization rule for unary operators is straightforward:

[UnaryOp *Expr*]
==

$$\begin{array}{l}\text{let } \$coree1 := (\llbracket\ Expr_1\ \rrbracket),\\ \quad \$e1 := \llbracket\ \$coree1\ \rrbracket_{\text{Optional\_Atomic\_Value(xs:double)}},\\ \text{return } \llbracket\ \text{UnaryOp; } \$e1\ \rrbracket_{\text{UnaryOp}}\end{array}$$

Core Grammar

There are no core grammar rules for arithmetic expressions as they are normalized to function calls.

Static Type Analysis

In the [XQuery 1.0 and XPath 2.0 Functions and Operators], type promotion rules are given for all the arithmetic operators, denoted by op:operation, and the result types of these operations. The following static semantics rules specifies the result types for all arithmetic operators when applied to specific fs:numeric types.

$$\frac{\text{statEnvs} \vdash Expr_1 : Type_1 \quad Type_1 <: \text{xs:decimal} \quad \text{statEnvs} \vdash Expr_2 : Type_2 \quad Type_2 <: \text{xs:decimal}}{\text{statEnvs} \vdash \text{op:operation}(Expr_1, Expr_2) : \text{xs:decimal}}$$

$$\frac{\text{statEnvs} \vdash Expr_1 : Type_1 \quad Type_1 <: \text{xs:float} \quad \text{statEnvs} \vdash Expr_2 : Type_2 \quad Type_2 <: \text{xs:float}}{\text{statEnvs} \vdash \text{op:operation}(Expr_1, Expr_2) : \text{xs:float}}$$

$$\frac{\text{statEnvs} \vdash Expr_1 : Type_1 \quad Type_1 <: \text{xs:double} \quad \text{statEnvs} \vdash Expr_2 : Type_2 \quad Type_2 <: \text{xs:double}}{\text{statEnvs} \vdash \text{op:operation}(Expr_1, Expr_2) : \text{xs:double}}$$

Ed. Note: MFF: Can the three rules above be factored?

Analogous static type rules are given for the unary arithmetic operators.

$$\frac{\text{statEnvs} \vdash Expr_1 : Type_1 \quad Type_1 <: \text{xs:decimal}}{\text{statEnvs} \vdash \text{op:operation}(Expr_1) : \text{xs:decimal}}$$

$$\frac{\text{statEnvs} \vdash Expr_1 : Type_1 \quad Type_1 <: \text{xs:float}}{\text{statEnvs} \vdash \text{op:operation}(Expr_1) : \text{xs:float}}$$

$$\frac{\text{statEnvs} \vdash Expr_1 : Type_1 \quad Type_1 <: \text{xs:double}}{\text{statEnvs} \vdash \text{op:operation}(Expr_1) : \text{xs:double}}$$

Dynamic Evaluation

The normalization rules map all arithmetic operators into core expressions, whose dynamic semantics is defined in other sections, therefore there are no dynamic semantics rules for arithmetic operators. The dynamic semantics rules for function calls given in **[4.2.4 Function Calls]** are applied to all the function calls

op:numeric-add, etc.

4.6 Comparison Expressions

Introduction

This section defines the semantics of [2.6 Comparison Expressions] in [XQuery 1.0: A Query Language for XML].

Comparison expressions allow two values to be compared. XQuery provides four kinds of comparison expressions, called value comparisons, general comparisons, node comparisons, and order comparisons.

[13]	ValueComp	::=	Expr ("eq" | "ne" | "lt" | "le" | "gt" | "ge") Expr
[12]	GeneralComp	::=	Expr ("=" | "!=" | "<" S | "<=" | ">" | ">=") Expr
[14]	NodeComp	::=	Expr ("==" | "!==") Expr
[15]	OrderComp	::=	Expr ("<<" | ">>" | "precedes" | "follows") Expr

4.6.1 Value Comparisons

Normalization

The value comparison equality operators "`eq`" and "`ne`" are defined on a large set of types.

$$\left[Expr_1 \text{ ValueEqOp } Expr_2 \right] ==$$

let $coree1 := ($\left[Expr_1 \right]$),
$coree2 := ($\left[Expr_2 \right]$),
$e1 := $\left[\$coree1 \right]_{\text{Optional\_Atomic\_Value}}$,
$e2 := $\left[\$coree2 \right]_{\text{Optional\_Atomic\_Value}}$
return
typeswitch ($e1) as $v1
case () return ()
case fs:numeric return
(typeswitch ($e2) as $v2
case () return ()
case fs:numeric return $\left[\$v1; \text{fs:numeric; ValueEqOp; } \$v2; \text{ fs:numeric} \right]_{\text{BinaryOp}}$
case fs:UnknownSimpleType return

$\left[\right.$ $v1; fs:numeric; ValueEqOp; (cast ($v2) as xs:double); fs:numeric $\left.\right]_{\text{BinaryOp}}$

default return dm:error())
case xs:boolean return
(typeswitch ($e2) as $v2
case () return ()
case xs:boolean return $\left[\right.$ $v1; xs:boolean; ValueEqOp; $v2; xs:boolean $\left.\right]_{\text{BinaryOp}}$
case fs:UnknownSimpleType return
$\left[\right.$ $v1; xs:boolean; ValueEqOp; (cast ($v2) as xs:boolean); xs:boolean $\left.\right]_{\text{BinaryOp}}$
default return dm:error())
case xs:string return
(typeswitch ($e2) as $v2
case () return ()
case xs:string return $\left[\right.$ $v1; xs:string; ValueEqOp; $v2; xs:string $\left.\right]_{\text{BinaryOp}}$
case fs:UnknownSimpleType return
$\left[\right.$ $v1; xs:string; ValueEqOp; (cast ($v2) as xs:string); xs:string $\left.\right]_{\text{BinaryOp}}$
default return dm:error())
case xs:date return
(typeswitch ($e2) as $v2
case () return ()
case xs:date return $\left[\right.$ $v1; xs:date; ValueEqOp; $v2; xs:date $\left.\right]_{\text{BinaryOp}}$
case fs:UnknownSimpleType return
$\left[\right.$ $v1; xs:date; ValueEqOp; (cast ($v2) as xs:date); xs:date $\left.\right]_{\text{BinaryOp}}$
default return dm:error())
case xs:time return
(typeswitch ($e2) as $v2
case () return ()
case xs:time return $\left[\right.$ $v1; xs:time; ValueEqOp; $v2; xs:time $\left.\right]_{\text{BinaryOp}}$
case fs:UnknownSimpleType return
$\left[\right.$ $v1; xs:time; ValueEqOp; (cast ($v2) as xs:time); xs:time $\left.\right]_{\text{BinaryOp}}$
default return dm:error())
case xs:dateTime return
(typeswitch ($e2) as $v2
case () return ()

case xs:dateTime return $[$ $v1; xs:dateTime; ValueEqOp; $v2; xs:dateTime $]_{BinaryOp}$

case fs:UnknownSimpleType return

$[$ $v1; xs:dateTime; ValueEqOp; (cast ($v2) as xs:dateTime); xs:dateTime $]_{BinaryOp}$

default return dm:error())

case xs:duration return

(typeswitch ($e2) as $v2

case () return ()

case xs:duration return $[$ $v1; xs:duration; ValueEqOp; $v2; xs:duration $]_{BinaryOp}$

case fs:UnknownSimpleType return

$[$ $v1; xs:duration; ValueEqOp; (cast ($v2) as xs:duration); xs:duration $]_{BinaryOp}$

default return dm:error())

case Gregorian return

(typeswitch ($e2) as $v2

case () return ()

case Gregorian return $[$ $v1; Gregorian; ValueEqOp; $v2; Gregorian $]_{BinaryOp}$

case fs:UnknownSimpleType return

$[$ $v1; Gregorian; ValueEqOp; (cast ($v2) as Gregorian); Gregorian $]_{BinaryOp}$

default return dm:error())

case xs:hexBinary return

(typeswitch ($e2) as $v2

case () return ()

case xs:hexBinary return $[$ $v1; xs:hexBinary; ValueEqOp; $v2; xs:hexBinary $]_{BinaryOp}$

case fs:UnknownSimpleType return

$[$ $v1; xs:hexBinary; ValueEqOp; (cast ($v2) as xs:hexBinary); xs:hexBinary $]_{BinaryOp}$

default return dm:error())

case xs:base64Binary return

(typeswitch ($e2) as $v2

case () return ()

case xs:base64Binary return $[$ $v1; xs:base64Binary; ValueEqOp; $x2; xs:base64Binary $]_{BinaryOp}$

case fs:UnknownSimpleType return

$\left[\right.$ $v1; xs:base64Binary; ValueEqOp; (cast ($v2) as xs:base64Binary); xs:base64Binary $\left.\right]_{\text{BinaryOp}}$
default return dm:error())
case xs:anyURI return
(typeswitch ($e2) as $v2
case () return ()
case xs:anyURI return $\left[\right.$ $v1; xs:anyURI; ValueEqOp; $v2; xs:anyURI $\left.\right]_{\text{BinaryOp}}$
case fs:UnknownSimpleType return
$\left[\right.$ $v1; xs:anyURI; ValueEqOp; (cast ($v2) as xs:anyURI); xs:anyURI $\left.\right]_{\text{BinaryOp}}$
default return dm:error())
case xs:QName return
(typeswitch ($e2) as $v2
case () return ()
case xs:QName return $\left[\right.$ $v1; xs:QName; ValueEqOp; $v2; xs:QName $\left.\right]_{\text{BinaryOp}}$
case fs:UnknownSimpleType return
$\left[\right.$ $v1; xsQName; ValueEqOp; (cast ($v2) as xs:QName) ; xs:QName $\left.\right]_{\text{BinaryOp}}$
default return dm:error())
case fs:UnknownSimpleType return
(typeswitch ($e2) as $v2
case () return ()
case fs:numeric return
$\left[\right.$ (cast ($v1) as xs:double); xs:double; ValueEqOp; $v2; xs:double; fs:numeric $\left.\right]_{\text{BinaryOp}}$
case xs:string return
$\left[\right.$ (cast ($v1) as xs:string); xs:string; ValueEqOp; $v2; xs:string $\left.\right]_{\text{BinaryOp}}$
case xs:date return
$\left[\right.$ (cast ($v1) as xs:date); xs:date; ValueEqOp; $v2; xs:date $\left.\right]_{\text{BinaryOp}}$
case xs:time return
$\left[\right.$ (cast ($v1) as xs:time); xs:time; ValueEqOp; $v2; xs:time $\left.\right]_{\text{BinaryOp}}$
case xs:dateTime return
$\left[\right.$ (cast ($v1) as xs:dateTime); xs:dateTime; ValueEqOp; $v2; xs:dateTime $\left.\right]_{\text{BinaryOp}}$
case xs:duration return
$\left[\right.$ (cast ($v1) as xs:duration); xs:duration; ValueEqOp; $v2; xs:duration $\left.\right]_{\text{BinaryOp}}$,
case fs:UnknownSimpleType return

$\llbracket$ (cast ($v1) as xs:string); xs:string; ValueEqOp; (cast ($v2) as xs:string); xs:string $\rrbracket_{\text{BinaryOp}}$
default return dm:error())
default return dm:error()

Ed. Note: MFF: The definition of equality operators could be factored by introducing another normalization function, which would be applied to the bodies of the cases. Is it clearer (albeit longer) to just enumerate all the cases? For now, they are enumerated.

$$\llbracket \mathit{Expr}_2\ \mathit{Type}_2\ \text{ValueEqOp;} \rrbracket_{\text{ValueOp}}$$
$$==$$

(typeswitch ($Expr_2$) as $v2
case () return ()
case return $\llbracket$ $v1; ValueEqOp; $v2; $\rrbracket_{\text{BinaryOp}}$
case fs:UnknownSimpleType return
$\llbracket$ $v1; $Type_2$ ValueEqOp; (cast ($v2) as $Type_2$); $v2 $\rrbracket_{\text{BinaryOp}}$
default return dm:error())

Normalization

The value comparison in-equality operators "`lt`", "`le`", "`gt`", and "`ge`" are defined on a smaller set of types than are the equality operators "`eq`" and "`ne`". For convenience, ValueInEqOp denotes "`lt`", "`le`", "`gt`", or "`ge`".

$$\llbracket \mathit{Expr}_1\ \text{ValueInEqOp}\ \mathit{Expr}_2 \rrbracket$$
$$==$$

let $coree1 := ($\llbracket$ $Expr_1$ $\rrbracket$),
$coree2 := ($\llbracket$ $Expr_2$ $\rrbracket$),
$e1 := $\llbracket$ $coree1 $\rrbracket_{\text{Optional\_Atomic\_Value}}$,
$e2 := $\llbracket$ $coree2 $\rrbracket_{\text{Optional\_Atomic\_Value}}$
return
typeswitch ($e1) as $v1
case () return ()
case fs:numeric return
(typeswitch ($e2) as $v2
case () return ()

case fs:numeric return $[$ $v1; fs:numeric; ValueInEqOp; $v2; fs:numeric $]_{BinaryOp}$

case fs:UnknownSimpleType return

$[$ $v1; fs:numeric; ValueInEqOp; (cast ($v2) as xs:double); fs:numeric $]_{BinaryOp}$

default return dm:error())

case xs:string return

(typeswitch ($e2) as $v2

case () return ()

case xs:string return $[$ $v1; xs:string; ValueInEqOp; $v2; xs:string $]_{BinaryOp}$

case fs:UnknownSimpleType return

$[$ $v1; xs:string; ValueInEqOp; (cast ($v2) as xs:string); xs:string $]_{BinaryOp}$

default return dm:error())

case xs:date return

(typeswitch ($e2) as $v2

case () return ()

case xs:date return $[$ $v1; xs:date; ValueInEqOp; $v2; xs:date $]_{BinaryOp}$

case fs:UnknownSimpleType return

$[$ $v1; xs:date; ValueInEqOp; (cast ($v2) as xs:date); xs:date $]_{BinaryOp}$

default return dm:error())

case xs:time return

(typeswitch ($e2) as $v2

case () return ()

case xs:time return $[$ $v1; xs:time; ValueInEqOp; $v2; xs:time $]_{BinaryOp}$

case fs:UnknownSimpleType return

$[$ $v1; xs:time; ValueInEqOp; (cast ($v2) as xs:time); xs:time $]_{BinaryOp}$

default return dm:error())

case xs:dateTime return

(typeswitch ($e2) as $v2

case () return ()

case xs:dateTime return $[$ $v1; xs:dateTime; ValueInEqOp; $v2; xs:dateTime $]_{BinaryOp}$

case fs:UnknownSimpleType return

$$[\ \$v1;\ xs:dateTime;\ ValueInEqOp;\ (cast\ (\$v2)\ as\ xs:dateTime);\ xs:dateTime\ ]_{BinaryOp}$$

default return dm:error())
case fs:UnknownSimpleType return
(typeswitch ($e2) as $v2
case () return ()
case fs:numeric return

$$[\ (cast\ (\$v1)\ as\ xs:double);\ xs:double;\ ValueInEqOp;\ \$v2;\ fs:numeric\ ]_{BinaryOp}$$

case xs:string return

$$[\ (cast\ (\$v1)\ as\ xs:string);\ xs:string;\ ValueInEqOp;\ \$v2;\ xs:string\ ]_{BinaryOp}$$

case xs:date return

$$[\ ValueInEqOp;\ xs:date;\ xs:date\ ]_{BinaryOp}(cast\ (\$v1)\ as\ xs:date,\ \$v2)$$

case xs:time return

$$[\ (cast\ (\$v1)\ as\ xs:time);\ xs:time;\ ValueInEqOp;\ \$v2;\ xs:time\ ]_{BinaryOp}$$

case xs:dateTime return

$$[\ (cast\ (\$v1)\ as\ xs:dateTime);\ xs:dateTime;\ ValueInEqOp;\ \$v2;\ xs:dateTime\ ]_{BinaryOp}$$

case xs:duration return

$$[\ (cast\ (\$v1)\ as\ xs:duration);\ xs:duration;\ ValueInEqOp;\ \$v2;\ xs:duration\ ]_{BinaryOp}$$

case fs:UnknownSimpleType return

$$[\ (cast\ (\$v1)\ as\ xs:string);\ xs:string;\ ValueInEqOp;\ (cast\ (\$v2)\ as\ xs:string);\ xs:string\ ]_{BinaryOp}$$

default return dm:error())
default return dm:error()

Core Grammar

There are no core grammar rules for value comparisons as they are normalized to function calls.

Static Type Analysis

There are no static type rules for the general comparison operators. They all have return type xs:boolean, as specified in [XQuery 1.0 and XPath 2.0 Functions and Operators].

Dynamic Evaluation

The normalization rules map all value comparison operators into core expressions, whose dynamic semantics is defined in other sections, therefore there are no dynamic semantics rules for value comparison operators. The dynamic semantics rules for function calls given in **[4.2.4 Function Calls]** are applied to all the function calls op:numeric-less-than, etc.

4.6.2 General Comparisons

Introduction

General comparisons are defined by adding existential semantics to 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`.

Notation

For convenience, GeneralOp denotes "=", "!=", "<", "", ">", or "".

The function $[\,]_{\text{ValueOp}}$ is defined by the following table:

GeneralOp	$[\text{GeneralOp}]_{\text{ValueOp}}$
=	eq
!=	ne
<	lt
<=	le
>	gt
>=	ge

Normalization

A general comparison expression is normalized by mapping it into an existentially quantified, value-comparison expression, which is normalized recursively.

$$[\,\textit{Expr}_1 \text{ GeneralOp } \textit{Expr}_2\,]$$
$$==$$
$$[\,\text{some \$v1 in } \textit{Expr}_1 \text{ satisfies (some \$v2 in } \textit{Expr}_2 \text{ satisfies } [\,\text{GeneralCompOp}\,]_{\text{ValueOp}}(\$v1, \$v2))\,]$$

Core Grammar

There are no core grammar rules for general comparisons as they are normalized to existentially quantified core expressions.

Static Type Analysis

There are no static type rules for the general comparison operators. The existentially quantified some expression always returns xs:boolean. Its static typing semantics is given in **[4.12 Quantified Expressions]**.

Dynamic Evaluation

The normalization rules map all general comparison operators into core expressions, whose dynamic semantics is defined in other sections, therefore there are no dynamic semantics rules for general comparison operators.

4.6.3 Node Comparisons

Notation

For convenience, NodeOp denotes "==" and "!==".

Normalization

The normalization rule for node comparison expressions checks that both operands are optional node values, otherwise generates an error. If both operands are nodes, it applies the operator specified by the $[\ ]_{\text{BinaryOp}}$ function.

$$[\![Expr_1 \text{ NodeOp } Expr_2]\!] \quad == \quad$$

```
let $e1 := ([[ Expr1 ]]),
    $e2 := ([[ Expr2 ]]),
return
typeswitch ($e1) as $v1
case () return
    (typeswitch ($e2) as $v2
    case node? return ()
    default return dm:error())
case node return
    (typeswitch ($e2) as $v2
    case () return ()
    case node return [[ $v1; node; NodeOp; $v2; node ]]BinaryOp
    default return dm:error())
default return dm:error()
```

Core Grammar

There are no core grammar rules for node comparisons as they are normalized to function calls.

Static Type Analysis

There are no static type rules for the node comparison operators.

Dynamic Evaluation

The normalization rules map the node comparison operators into core expressions, whose dynamic semantics is defined in other sections, therefore there are no dynamic semantics rules for node comparison operators.

4.6.4 Order Comparisons

Notation

For convenience, OrderOp denotes "follow", "precedes", "<<", and ">>".

Normalization

The normalization rule for order comparison expressions checks that both operands are optional node values, otherwise generates an error. If both operands are nodes, it applies the operator specified by the $[\ ]_{\text{BinaryOp}}$ function.

$$[\![Expr_1 \text{ OrderOp } Expr_2]\!]$$
$$==$$

```
let $e1 := ([ Expr1 ]),
    $e2 := ([ Expr2 ]),
return
typeswitch ($e1) as $v1
case () return
    (typeswitch ($e2) as $v2
    case node? return ()
    default return dm:error())
case node return
    (typeswitch ($e2) as $v2
    case () return ()
    case node return [ $v1; node; OrderOp; $v2; node ]BinaryOp
    default return dm:error())
default return dm:error()
```

Core Grammar

There are no core grammar rules for order comparisons as they are normalized to function calls.

Static Type Analysis

There are no static type rules for the order comparison operators.

Dynamic Evaluation

The normalization rules map the order comparison operators into core expressions, whose dynamic semantics is defined in other sections, therefore there are no dynamic semantics rules for order comparison operators.

4.7 Logical Expressions

Introduction

This section defines the semantics of [2.7 Logical Expressions] in [XQuery 1.0: A Query Language for XML].

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`.

[6] OrExpr ::= Expr "or" Expr
[7] AndExpr ::= Expr "and" Expr

Notation

The first step in evaluating a logical expression is to reduce each of its operands to an **effective boolean value**. The function $[\ ]_{\text{Effective\_Boolean\_Value}}$ takes an expression and normalizes it into an effective boolean value. The conditional expression in the "default return" clause below guarantees that in an arbitrary sequence of items, if at least one value is a node, then the boolean expression evaluates to true.

$$[\![Expr]\!]_{\text{Effective\_Boolean\_Value}} =$$
==
typeswitch (*Expr*) as $v
case () return xf:false()
case xs:boolean return $v
default return
if (xf:length($[\![\$v/self::node()]\!]_{\text{Path}}$) >= 1) then xf:true()
else $[\![\$v]\!]_{\text{Type\_Exception\_Opt\_Atomic(boolean)}}$

Ed. Note: The semantics of Boolean node tests over sequences is still an open issue. See [**Issue-0131: Boolean node test and sequences**].

Normalization

The normalization rules for "and" and "`or`" first get the effective boolean value of each argument, then apply the appropriate operand.

$$[\![Expr_1 \text{ and } Expr_2]\!]$$
==
let $e1 := $[\![Expr_1]\!]_{\text{Effective\_Boolean\_Value}}$,
$e2 := $[\![Expr_2]\!]_{\text{Effective\_Boolean\_Value}}$,
return op:boolean-and($e1, $e2)

$$[\![Expr_1 \texttt{ or } Expr_2]\!]$$
==

$$\begin{aligned}&\text{let } \$e1 := \left[\ Expr_1\ \right]_{\text{Effective\_Boolean\_Value}},\\ &\quad\ \$e2 := \left[\ Expr_2\ \right]_{\text{Effective\_Boolean\_Value}},\\ &\text{return op:boolean-or(\$e1, \$e2)}\end{aligned}$$

Core Grammar

There are no core grammar rules for logical expressions as they are normalized to function calls.

Static Type Analysis

There are no static type rules for the logical comparison operators. They both have return type xs:boolean, as specified in [XQuery 1.0 and XPath 2.0 Functions and Operators].

Dynamic Evaluation

The normalization rules map the logical comparison operators into core expressions, whose dynamic semantics is defined in other sections, therefore there are no dynamic semantics rules for logical comparison operators.

4.8 Constructors

> **Ed. Note: Status:** This section is still draft and needs further revision. Revision of the section is pending agreement about the semantics of element and attribute constructors. See **[Issue-0110: Semantics of element and attribute constructors]**

This section defines the semantics of [2.8 Constructors] in [XQuery 1.0: A Query Language for XML].

XQuery supports two forms of constructors: a "literal" form that follows the XML syntax, and element and attribute constructors that can be used to construct stand-alone elements and attributes, possibly with a computed name.

[24]	Constructor	::=	ElementConstructor | ComputedElementConstructor | ComputedAttributeConstructor
[57]	ElementConstructor	::=	"<" QName AttributeList ("/>" | (">" ElementContent* "</" QName ">"))
[63]	ElementContent	::=	Char | "{{" | "}}" | ElementConstructor | EnclosedExpr | CdataSection | CharRef | PredefinedEntityRef | XmlComment | XmlProcessingInstruction | ComputedElementConstructor | ComputedAttributeConstructor

[64]	AttributeList	::=	(QName "=" (AttributeValue | EnclosedExpr))*
[65]	AttributeValue	::=	(["] AttributeValueContent* ["]) | (['] AttributeValueContent* ['])
[66]	AttributeValueContent	::=	Char | CharRef | "{{" | "}}" | EnclosedExpr | PredefinedEntityRef
[67]	EnclosedExpr	::=	"{" ExprSequence "}"

4.8.1 Element Constructors

Introduction

We define the semantics of the literal form of element constructors by normalizing it to a computed element constructor.

Normalization

> **Ed. Note: Status:**The specification of element constructor normalization is pending adoption of a proposal on the treatment of whitespace in XQuery expressions and of a proposal on the semantics of element constructors. See **[Issue-0110: Semantics of element and attribute constructors]**.

4.8.2 Computed Element and Attribute Constructors

[58]	ComputedElementConstructor	::=	"element" (QName | EnclosedExpr) "{" ExprSequence? "}"
[59]	ComputedAttributeConstructor	::=	"attribute" (QName | EnclosedExpr) "{" ExprSequence? "}"

Normalization

Computed element and attribute expressions are normalized by mapping their sub-expressions:

$$\left[\text{element } (\mathit{QName} \mid \{\ \mathit{Expr}\ \})\ \{\ \mathit{ExprSequence}\ \}\right]$$
$$==$$
$$\text{element } (\left[\mathit{QName}\right] \mid \{\ \left[\mathit{Expr}\right]\ \})\ \{\ \left[\mathit{ExprSequence}\right]\}$$

$$\left[\text{attribute } (\mathit{QName} \mid \{\ \mathit{Expr}\ \})\ \{\ \mathit{ExprSequence}\ \}\right]$$
$$==$$
$$\text{attribute } (\left[\mathit{QName}\right] \mid \{\ \left[\mathit{Expr}\right]\ \})\ \{\ \left[\mathit{ExprSequence}\right]\}$$

Core Grammar

The core grammar rules for computed constructors are:

[58] ComputedElementConstructor ::= "element" (QName | EnclosedExpr) "{" ExprSequence? "}"
[59] ComputedAttributeConstructor ::= "attribute" (QName | EnclosedExpr) "{" ExprSequence? "}"

Static Type Analysis

The normalization rules leave us with only a the operator form of element (resp. attribute) constructor to handle. The element (resp. attribute) operator still has two form: one in which a *QName* is supplied as the element name, and one in which a computed expression is supplied. In the latter case, we are unable to provide anything other than a wildcard type, since the element name cannot be known until runtime.

Note that when the *QName* form is used, the resulting type is a *constructed* element type. This means that the element type is consistent with its contents (as determined by static analysis), but it does not necessarily bear any relation to any declared element type of the same name. In other words, there is no schema validation implied here. Moreover, neither XQuery nor this document have yet specified how validation can be invoked, other than with the treat as operator.

> **Ed. Note:** DD: the above is a necessary consequence of our current evaluation model. One variation that I can think of is that the static rules could try looking up the element name in ; if the element name is found, then the associated, named type is used (and the contents are required to match). If the named element type is not found, then the constructed type is used, as before. Thus, in this scenario, validation would be done whenever an element with a "global" name was created. See also comments on the impact of (non-)validation on element construction, in the next section.

$$\frac{\begin{array}{c}\text{statEnvs} \text{ |- } \text{expand}(\textit{QName}) = \textit{qname} \\ \text{statEnvs} \text{ |- } \textit{ExprSequence} : \textit{Type}_1 \\ \textit{Type}_1 \text{ <: } (\text{ xs:AnyAttribute*, (xs:AnySimpleType | (xs:AnyElement | xs:string)*))}\end{array}}{\text{statEnvs} \text{ |- element } \textit{QName} \text{ \{ } \textit{ExprSequence} \text{ \} : element } \textit{qname} \text{ \{ } \textit{Type}_1 \text{ \}}}$$

$$\frac{\begin{array}{c}\text{statEnvs} \text{ |- } \textit{Expr} : \text{xs:QName} \\ \text{statEnvs} \text{ |- } \textit{ExprSequence} : \textit{Type}_1 \\ \textit{Type}_1 \text{ <: } (\text{ xs:AnyAttribute*, (xs:AnySimpleType | (xs:AnyElement | xs:string)*))}\end{array}}{\text{statEnvs |- element \{ } \textit{Expr} \text{ \} \{ } \textit{ExprSequence} \text{ \} : element * \{ } \textit{Type} \text{ \}}}$$

$$\frac{\begin{array}{c}\text{statEnvs} \text{ |- } \textit{Expr} : \text{xs:QName} \\ \text{statEnvs} \text{ |- } \textit{ExprSequence} : \textit{Type}_1 \\ \textit{Type}_1 \text{ <: xs:AnySimpleType}\end{array}}{\text{statEnvs |- attribute \{ } \textit{Expr} \text{ \} \{ } \textit{ExprSequence} \text{ \} : attribute * \{ } \textit{Type} \text{ \}}}$$

$$\frac{\begin{array}{c}\text{statEnvs} \text{ |- } \text{expand}(\textit{QName}) = \textit{qname} \\ \text{statEnvs} \text{ |- } \textit{ExprSequence} : \textit{Type}_1 \\ \textit{Type}_1 \text{ <: xs:AnySimpleType}\end{array}}{\text{statEnvs} \text{ |- attribute } \textit{QName} \text{ \{ } \textit{ExprSequence} \text{ \} : attribute } \textit{qname} \text{ \{ } \textit{Type}_1 \text{ \}}}$$

Dynamic Evaluation

The following rule constructs an element (resp. attribute) from its name and children sub-expressions.

dynEnvs |- *NameSpec* => *qname*
dynEnvs |- *ExprSequence* => ($anyAttribute_1$, ... $anyAttribute_n$, $value_1$, ..., $value_n$)

---

dynEnvs |- element *NameSpec* { *ExprSequence* } =>

`dm:element-simple-node`(*qname*,
`xf:empty-sequence`(),
($anyAttribute_1$, ..., $anyAttribute_n$),
($value_1$, ..., $value_n$),
`xf:empty-sequence`())

dynEnvs |- *NameSpec* => *qname*
dynEnvs |- *ExprSequence* => ($value_1$, ..., $value_n$)

---

dynEnvs |- attribute *NameSpec* { *ExprSequence* } =>

`dm:attribute-simple-node`(*qname*,
($value_1$, ..., $value_n$),
`xf:empty-sequence`())

Note that the element constructor `dm:element-simple-node` functions by making copies of any Nodes in its arguments; therefore the result of element construction is a completely "new" element.

Ed. Note: DD: There are several issues with element construction:

- We do not supply either namespaces or schema-components to the constructor. We cannot do these things because of the bottom-up nature of element construction: we do not, in general, know either the namespaces in scope or the validation-associated schema type until this element has been "seated" in some containing element (and so on recursively).

  There is a possible solution to this chicken-and-egg problem, however: because the element constructor makes copies of its children, it could be the responsibility of the element *constructor* to "fill in" the values for namespaces-in-scope and schema-component on each newly-copied child (recursively), based on information provided for the node. In this scenario, these fields would remain "blank" until some appropriate activity caused a schema component to become associated with a node, etc.

  One implication of this scheme would be that the "value" of elements could change as they are copied into a new containing element. For example, defaulted attributes could be added. Possibly the interpretation of data values would change as well, e.g. a data value supplied as a string could be re-interpreted as a number.

- We do not handle any special treatment of xmlns, etc. that would be needed to associate namespaces with elements.

- Even if/when schema components are available, it is not clear when or how defaulted attributes and/or elements are created.

- The conversion of atomic values to text nodes, plus the lack of schema components, means that the data

model must necessarily lose type information about element contents. Thus the data model function `dm:typed-value` cannot do what we need it to do. Possibly the text node constructor `dm:text-node` could keep track of the types of the atomic values used to create the text node?

4.8.3 Other Constructors and Comments

[60]	CdataSection	::=	"<![CDATA[" Char* "]]>"
[61]	XmlProcessingInstruction	::=	"<?" PITarget Char* "?>"
[62]	XmlComment	::=	"<!--" Char* "-->"

XQuery currently does not define how comments and processing instructions are created.

4.9 FLWR Expressions

Introduction

This section defines the semantics of [2.9 FLWR Expressions] in [XQuery 1.0: A Query Language for XML].

XQuery provides a FLWR expression for iteration, for binding variables to intermediate results, and to filter bound variables based on a predicate.

A *FLWRExpr* in XQuery 1.0 consists of a sequence of *ForClause*s, *LetClause*s, and an optional *WhereClause*, followed by a return clause, as described by the following grammar productions.

[8]	FLWRExpr	::=	(ForClause | LetClause)+ WhereClause? "return" Expr
[27]	ForClause	::=	"for" Variable "in" Expr ("," Variable "in" Expr)*
[28]	LetClause	::=	"let" Variable ":=" Expr ("," Variable ":=" Expr)*
[29]	WhereClause	::=	"where" Expr

4.9.1 FLWR expressions

Notation

Individual FLWR clauses are normalized by means of the auxiliary normalization rules:

$$\left[FLWRClause \right]_{\text{flwr\_clause}}(Expr)$$

Where *FLWRClause* can be any either a *ForClause*, a *LetClause*, or a *WhereClause*:

[100]	FLWRClause	::=	ForClause | LetClause | WhereClause

Note that, as is, this auxiliary rule normalizes a fragment of the FLWR expression, while taking the remainder of the expression (in *Expr*) as an additional parameter.

Normalization

Full FLWR expressions are normalized to nested core expressions using two sets of normalization rules. Note that some of the rules also accepts ungrammatical *FLWRExpr*s such as "where $Expr_1$ return $Expr_2$". This does not matter, as normalization is always applied on parsed XQuery expressions, and ungrammatical *FLWRExpr*s would be rejected by the parser beforehand.

The first set of rules is applied on a full FLWR expression, splitting it at the clause level, then applying further normalization on each separate clause.

$$\begin{gathered}\left[\ (\mathit{ForClause} \mid \mathit{LetClause} \mid \mathit{WhereClause})\ \mathit{FLWRExpr}\ \right] \\ == \\ \left[(\mathit{ForClause} \mid \mathit{LetClause} \mid \mathit{WhereClause})\right]_{\text{flwr\_clause}}(\left[\mathit{FLWRExpr}\right])\end{gathered}$$

$$\begin{gathered}\left[\ (\mathit{ForClause} \mid \mathit{LetClause} \mid \mathit{WhereClause})\ \text{return}\ \mathit{Expr}\ \right] \\ == \\ \left[(\mathit{ForClause} \mid \mathit{LetClause} \mid \mathit{WhereClause})\right]_{\text{flwr\_clause}}(\left[\mathit{Expr}\right])\end{gathered}$$

Then each FLWR clause is normalized separately. A *ForClause* may bind more than one variable, whereas a for expression in the XQuery core binds and iterates over only one variable. Therefore, a *ForClause* is normalized to nested for expressions:

$$\begin{gathered}\left[\ \text{for}\ \mathit{Variable}_1\ \text{in}\ \mathit{Expr}_1,\ldots,\mathit{Variable}_\text{n}\ \text{in}\ \mathit{Expr}_\text{n}\ \right]_{\text{flwr\_clause}}(\mathit{Expr}) \\ == \\ \text{for}\ \mathit{Variable}_1\ \text{in}\ \left[\mathit{Expr}_1\ \right]\ \text{return} \\ \cdots \\ \text{for}\ \mathit{Variable}_\text{n}\ \text{in}\ \left[\ \mathit{Expr}_\text{n}\ \right]\ \text{return}\ \mathit{Expr}\end{gathered}$$

Note that the additional *Expr* parameter of the auxiliary normalization rule is used as the final return expression.

Likewise, a *LetClause* clause is normalized to nested let expressions:

$$\begin{gathered}\left[\ \text{let}\ \mathit{Variable}_1 := \mathit{Expr}_1, \cdots,\mathit{Variable}_\text{n} := \mathit{Expr}_\text{n}\right]_{\text{flwr\_clause}}(\mathit{Expr}) \\ == \\ \text{let}\ \mathit{Variable}_1 := \left[\mathit{Expr}_1\ \right]\ \text{return} \\ \cdots \\ \text{let}\ \mathit{Variable}_\text{n} := \left[\mathit{Expr}_\text{n}\right]\ \text{return}\ \mathit{Expr}\end{gathered}$$

A *WhereClause* is normalized to an *IfExpr*, with the else-branch returning the empty list:

$$\begin{gathered}\left[\ \text{where}\ \mathit{Expr}_1\right]_{\text{flwr\_clause}}(\mathit{Expr}) \\ ==\end{gathered}$$

if ($\left[Expr_1 \right]$) then *Expr* else ()

Example

The following simple example illustrates, how a *FLWRExpr* is normalized. The FLWR expressions is used to iterate over two collections, binding variables `$i` and `$j` to items in these collections. It uses a `let` clause to binds the local variable `$k` to the sum of both numbers, and a where clause to selects those numbers that have only a sum equal to or greater than the integer `5`.

```
for $i in (1, 2),
    $j in (3, 4)
let $k := $i + $j
where $k >= 5
return
  <tuple>
     <i> { $i } </i>
     <j> { $j } </j>
  </tuple>
```

Through the first set of rules, this is normalized to (except for the operators and element constructor which are not treated here):

```
for $i in (1, 2) return
  for $j in (3, 4) return
    let $k := $i + $j return
      if ($k >= 5) then
        <tuple>
          <i> { $i } </i>
          <j> { $j } </j>
        </tuple>
      else
        ()
```

For each binding of `$i` to an item in the sequence `(1 , 2)` the inner for expression iterates over the sequence `(3 , 4)` to produce tuples ordered by the ordering of the outer sequence and then by the ordering of the inner sequence. We see in the rest of the section, how this core expression results in the following document fragment:

```
(<tuple>
    <i>1</i>
    <j>4</j>
 </tuple>,
 <tuple>
    <i>2</i>
    <j>3</j>
 </tuple>,
 <tuple>
```

```
    <i>2</i>
    <j>4</j>
  </tuple>)
```

with the static type:

```
element tuple {
  element i { xs:decimal },
  element j { xs:decimal }
}*
```

4.9.2 For expression

Core Grammar

After normalization single for expressions are described by the following core grammar production.

[0] ForExpr ::= ForClause "return" Expr

Static Type Analysis

A single for expression is typed as follows: First $Type_1$ of the iteration expression $Expr_1$ is inferred. Then the prime type of $Type_1$ - prime($Type_1$) - is determined. This is a choice of all item types in $Type_1$ (see also **[3.4 Prime types]**). With the variable component of the static environment statEnvs extended with $Variable_1$ of type prime($Type_1$), the type $Type_2$ of $Expr_2$ is inferred. Because the for expression iterates over the result of $Expr_1$, the final type of the iteration is $Type_2$ multiplied with the possible number of items in $Type_1$ (one, `?`, `*`, or `+`). This number is determined by the auxiliary type-function quantifier($Type_1$).

$$\frac{\text{statEnvs} \vdash Expr_1 : Type_1 \qquad \text{statEnvs}\,[\,\text{varType}(Variable_1 : \text{prime}(Type_1))\,] \vdash Expr_2 : Type_2}{\text{statEnvs} \vdash \text{for } Variable_1 \text{ in } Expr_1 \text{ return } Expr_2 : Type_2 \cdot \text{quantifier}(Type_1)}$$

Example

For example, if `$example` is bound to the sequence `(<one/> , <two/> , <three/>)` of type `element one {}, element two {}, element three {}`, then the query

```
for $s in $example
return <out> {$s} </out>
```

is typed as follows:

```
(1) prime(element one {}, element two {}, element three {}) =
    element one {} | element two {} | element three {}
(2) quantifier(element one {}, element two {}, element three {}) = +
(3) $s : element one {} | element two {} | element three {}
(4) <out> {$s} </out> :
    element out {element one {} | element two {} | element three {}}
(5) result-type :
    element out {element one {} | element two {} | element three {}}+
```

This result-type is not the most specific type possible. It does not take into account the order of elements in the input type, and it forgets about the individual and overall number of elements in the input type. The most specific type possible is: `element out {element one {}}, element out {element two {}}, element out {element three {}}`. However, inferring such a specific type for arbitrary input types and arbitrary return clauses requires significantly more complicated type inference rules. In addition, if put into the context of an element, the specific type violates the "consistent element restriction" of XML schema, which requires that an element must have a unique content model within a particular context.

Dynamic Evaluation

The evaluation of a for expression distinguishes two cases: If the iteration expression $Expr_1$ evaluates to the empty sequence, then the entire expression evaluates to the empty sequence.

$$\frac{\text{dynEnvs} \text{ |- } Expr_1 \text{ => } value \qquad \text{empty}(value)}{\text{dynEnvs} \text{ |- for } Variable_1 \text{ in } Expr_1 \text{ return } Expr_2 \text{ => dm:empty-sequence()}}$$

Otherwise, the iteration expression $Expr_1$, is evaluated to produce the sequence $itemValue_1$, ..., $itemValue_n$. Then for each item $itemValue_i$ in this sequence, the body of the for expression $Expr_2$ is evaluated in the environment dynEnvs extended with $Variable_1$ bound to $itemValue_i$. This produces values $value_i$, which are concatenated to produce the result sequence.

$$\frac{\begin{array}{c}\text{dynEnvs} \text{ |- } Expr_1 \text{ => } itemValue_1 \text{ ,..., } itemValue_n \\ \text{dynEnvs [varValue}(Variable_1 \text{ |-> } itemValue_1) \text{] |- } Expr_2 \text{ => } value_1 \\ \cdots \\ \text{dynEnvs [varValue}(Variable_1 \text{ |-> } itemValue_1) \text{] |- } Expr_2 \text{ => } value_n\end{array}}{\text{dynEnvs} \text{ |- for } Variable_1 \text{ in } Expr_1 \text{ return } Expr_2 \text{ => op:concatenate}(value_1 \text{ ,..., } value_n)}$$

> **Ed. Note:** The dynamic semantics of for could be better defined without the use of ⋯ and using recursion. See **[Issue-0134: Should we define for with head and tail?]**.

Example

Note that even if the expression in the return clause can result in a sequence, sequences are never nested in the XQuery data model. For instance, in the following for expression:

```
for $i in (1,2)
  return (<i> {$i} </i>, <negi> {-$i} </negi>)
```

each iteration in the for results in a sequence of two elements, which are then concatenated and flattened in the resulting sequence (through the op:concatenate function):

```
(<i>1</i>,
 <negi>-1</negi>,
 <i>2</i>,
 <negi>-2</negi>,
```

4.9.3 Let expression

Core Grammar

After normalization single let expressions are described by the following core grammar production.

[0] LetExpr ::= LetClause "return" Expr

Static Type Analysis

A let expression extends the type environment statEnvs with $Variable_1$ of type $Type_1$ inferred from $Expr_1$, and infers the type of $Expr_2$ in the extended environment to produce the result type $Type_2$.

$$\frac{\text{statEnvs} \vdash Expr_1 : Type_1 \qquad \text{statEnvs}\,[\,\text{varType}(Variable_1 : Type_1)\,] \vdash Expr_2 : Type_2}{\text{statEnvs} \vdash \text{let } Variable_1 := Expr_1 \text{ return } Expr_2 : Type_2}$$

Dynamic Evaluation

A let expression extends the dynamic environment dynEnvs with *Variable* bound to $value_1$ returned by $Expr_1$, and evaluates $Expr_2$ in the extended environment to produce $value_2$.

$$\frac{\text{dynEnvs} \vdash Expr_1 \Rightarrow value_1 \qquad \text{dynEnvs}\,[\,\text{varValue}(Variable_1 \mapsto itemValue_1)\,] \vdash Expr_2 \Rightarrow value_2}{\text{dynEnvs} \vdash \text{let } Variable_1 := Expr_1 \text{ return } Expr_2 \Rightarrow value_2}$$

Example

Note the use of the environment discipline to define the scope of each variables. For instance, in the following nested let expression:

```
let $k := 5 return
  let $k := $k + 1 return
    $k+1
```

the outermost let expression binds variable $k to the integer 5 in the environment, then the expression `$k+1` is computed, yielding value 6, to which the second variable $k is bound. The expression then results in the final integer 7.

4.10 Sorting Expressions

> **Ed. Note: Status:** This section is still incomplete and needs further revision. This revision is pending agreement about the semantics of sorting.See **[Issue-0109: Semantics of sortby]**.

Introduction

This section defines the semantics of [2.10 Sorting Expressions] in [XQuery 1.0: A Query Language for XML]. A sorting expression provides a way to specify the order of items in a sequence.

[5] SortExpr ::= Expr "stable"? "sortby" "(" SortSpecList ")"
[26] SortSpecList ::= Expr ("ascending" | "descending")? ("," SortSpecList)?

Normalization

In XQuery, the specification of ascending/descending is optional. We normalize missing sort order to

"ascending".

$$[\ Expr\ (,\ \text{SortSpecList})?\ ] \quad == \quad [\ Expr\ ]\ \text{ascending}\ (,\ [\ \text{SortSpecList}\ ])?$$

Core Grammar

The core grammar rule for the sort expression is:

[5] SortExpr ::= Expr "stable"? "sortby" "(" SortSpecList ")"

Static Type Analysis

The `sortby` expression returns prime(*Type*) · quantifier(*Type*) of its input type *Type* (see also **[3.4 Prime types]**).

$$\frac{\text{statEnvs} \vdash Expr : Type}{\text{statEnvs} \vdash Expr\ \texttt{sortby}\ SortSpecList : \text{prime}(Type) \cdot \text{quantifier}(Type)}$$

> **Ed. Note:** (MF) / Oct 23/2000: This definition assumes that the equality operator on $Type_2$ is defined. An alternative is requiring $Expr_2$ to have fs:atomic, but that seems too restrictive.
>
> **Ed. Note:** DD: My version is even worse: it doesn't even refer to the types of the SortSpecList; it is true this ought to check the types of the sortby expressions for the existence of an ordering relation. Also: we need to define how to sort nodes in document order.
>
> **Ed. Note:** Peter: yes indeed, this needs to be fixed. We need (a) a core xf:sort (with only one criterion) and appropriate more specific static and dynamic semantics, (b) a normalization of the surface sortby (with many criteria) to the core xf:sort. Maybe we should tie this to resolution of Dana's issue with sortby.

Dynamic Evaluation

The dynamic semantics of the sortby operator has not been defined. See **[Issue-0109: Semantics of sortby]**.

4.11 Conditional Expressions

Introduction

This section defines the semantics of [2.11 Conditional Expressions] in [XQuery 1.0: A Query Language for XML]. A conditional expression supports conditional evaluation of one of two expressions.

[11] IfExpr ::= "if" "(" Expr ")" "then" Expr "else" Expr

Normalization

$$\left[\text{if } (Expr_1) \text{ then } Expr_2 \text{ else } Expr_3\right]$$

==

$$\text{let \$fs:new} := \left[Expr_1 \right]_{\text{Effective\_Boolean\_Value}},$$
$$\text{return if (\$fs:new) then } \left[Expr_2\right] \text{ else } \left[Expr_3\right]$$

where $fs:new is a newly created variable that does not appear in the rest of the query.

Core Grammar

The core grammar rule for the conditional expression is:

[11] IfExpr ::= "if" "(" Expr ")" "then" Expr "else" Expr

Static Type Analysis

$$\frac{\text{statEnvs} \vdash Expr_1 : \text{xs:boolean} \quad \text{statEnvs} \vdash Expr_2 : Type_2 \quad \text{statEnvs} \vdash Expr_3 : Type_3}{\text{statEnvs} \vdash \texttt{if}\ (Expr_1)\ \texttt{then}\ Expr_2\ \texttt{else}\ Expr_3 : (Type_2 \mid Type_3)}$$

Dynamic Evaluation

If the conditional's boolean expression $Expr_1$ evaluates to true, $Expr_2$ is evaluated and its value is produced. If the conditional's boolean expression evaluates to false, $Expr_3$ is evaluated and its value is produced. Note that the existence of two separate evaluation rules ensures that only one branch of the conditional is evaluated.

> **Ed. Note:** DD: actually, I would like that last sentence to be true, but I don't think it is: there is nothing that I know of in the semantics of evaluation rules that says that the clauses must be evaluated left-to-right, so currently nothing stops an implementation from evaluating the body expressions first. Even if this is a functional language generally, I am sure there will be non-functional side-effects in implementation-dependent extensions to the language, and I think we must find a way to formally state that bodies of conditionals do not get executed if their test fails.

$$\frac{\text{dynEnvs} \vdash Expr_1 \Rightarrow \text{true} \quad \text{dynEnvs} \vdash Expr_2 \Rightarrow value_2}{\text{dynEnvs} \vdash \text{if } Expr_1 \text{ then } Expr_2 \text{ else } Expr_3 \Rightarrow value_2}$$

$$\frac{\text{dynEnvs} \vdash Expr_1 \Rightarrow \text{false} \quad \text{dynEnvs} \vdash Expr_3 \Rightarrow value_3}{\text{dynEnvs} \vdash \texttt{if}\ Expr_1\ \texttt{then} Expr_2\ \texttt{else} Expr_3 \Rightarrow value_3}$$

4.12 Quantified Expressions

Introduction

This section defines the semantics of [2.12 Quantified Expressions] in [XQuery 1.0: A Query Language for XML].

XQuery defines two quantification expressions:

[9] QuantifiedExpr ::= ("some" | "every") Variable "in" Expr "satisfies" Expr

Normalization

The quantification expressions are entirely normalized into other core expressions in the following normalization rules.

$$\Big[\text{some } \textit{Variable} \text{ in } \textit{Expr}_1 \text{ satisfies } \textit{Expr}_2\Big] ==$$

```
xf:not ( xf:empty(
  for Variable in [Expr1] return
   if ( [ Expr2 ]Effective_Boolean_Value ) then 1
   else ()
  ))
```

$$\Big[\text{every } \textit{Variable} \text{ in } \textit{Expr}_1 \text{ satisfies } \textit{Expr}_2\Big] ==$$

```
xf:empty(
  for Variable in [Expr1] return
   if ( xf:not( [ Expr2 ]Effective_Boolean_Value) ) then 1
   else ()
  )
```

Core Grammar

There are no core grammar rules for quantified expressions as they are normalized to other core expressions.

Dynamic Evaluation

There are no additional dynamic evaluation rules for the quantified expressions.

Static Type Analysis

There are no additional static type rules for the quantified expressions.

4.13 Datatypes

Introduction

This section defines the semantics of [2.13 Datatypes] in [XQuery 1.0: A Query Language for XML].

Datatypes can be used in the language to refer to a type declared in the query prolog (see **[5 The Query Prolog]**). Datatypes are used to declare the type of function parameters and in several kinds of XQuery expressions. We first describe the semantics of Datatypes with respect to the XQuery type system, then describe the semantics of expressions on Datatypes.

4.13.1 Referring to Datatypes

Introduction

The syntax of Datatypes is described by the following grammar productions.

[53]	Datatype	::=	(("element" "of" "type" QName) | DTKind | "node" | SimpleType | "item") OccurrenceIndicator
[54]	DTKind	::=	("element" | "attribute") QName?
[56]	OccurrenceIndicator	::=	("*" | "+" | "?")?

We give the semantics of Datatypes by means of normalization rules from Datatypes to the XQuery type system (see **[3 The XQuery Type System]**). We then describe the dynamic and static semantics of expressions involving datatypes in terms of operations on the XQuery type system.

> **Ed. Note:** Note that normalization on Datatypes does not occur during the normalization phase, but whenever a dynamic or static rule requires it. The reason for that deviation from the processing model is that the result of Datatype normalization is not part of the XQuery syntax (See issue [**Issue-0089: Syntax for types in XQuery**]). Datatype normalization is the only occurrence of such a deviation in the formal semantics.

Notation

To define the semantics of Datatypes, we make use of the following auxiliary normalization rule. The notation:

$$[\![\mathit{Datatype}]\!]_{\text{Datatype}} == \mathit{Type}$$

specifies that *Datatype* is equivalent to *Type*, in the XQuery type system.

We also rely of the following additional production, which facilitates the specification of the normalization rules for Datatypes.

[99]	DTComponent	::=	("element" "of" "type" QName) | DTKind | "node" | SimpleType | "item"

Normalization

OccurenceIndicators are left unchanged when normalizing Datatypes into XQuery types. Each kind of

Datatype component is normalized separately into the XQuery type system.

$$[\text{DTComponent OccurrenceIndicator}]_{\text{Datatype}} == [\text{DTComponent}]_{\text{Datatype}}\ \text{OccurrenceIndicator}$$

The Datatype component "element of type *QName*" can be used to refer to any element, (i.e., with any *qname*), whose type is the globally defined type *QName*. Note that the following mapping rule uses the type environment to make sure the global type exists, and a wildcard name for the resulting element.

If statEnvs.varType(*QName*) => *Type*, then

$$[\text{element of type } \mathit{QName}]_{\text{Datatype}} == \text{element * \{ type } \mathit{QName} \text{ \}}$$

otherwise it is an error.

The Datatype component DTKind can be used to refer to a globally defined element or attribute. In case the name of the element or attribute is missing, this means any element or attribute is allowed. Note that the following mapping rules use the type environment to make sure the global element or attribute exists, and a wildcard type in case the name of the element or attribute is missing.

If statEnvs.elemDecl(*QName*) => *Type*, then

$$[\text{element } \mathit{QName}]_{\text{Datatype}} == \text{element } \mathit{QName}$$

otherwise it is an error.

If statEnvs.attrDecl(*QName*) => *Type*, then

$$[\text{attribute } \mathit{QName}]_{\text{Datatype}} == \text{attribute } \mathit{QName}$$

otherwise it is an error.

$$[\text{element}]_{\text{Datatype}} == \text{xs:AnyElement}$$

$$[\text{attribute}]_{\text{Datatype}}$$

==

xs:AnyAttribute

The Datatype components "node" and "item" correspond to wildcard types. `node` indicates that any node is allowed, and `item` indicates that any node or value is allowed. The following mapping rules make use of the corresponding wildcard types.

$$[\text{node}]_{\text{Datatype}}$$

==

xs:AnyNode

$$[\text{item}]_{\text{Datatype}}$$

==

xs:AnyItem

The Datatype component SimpleType is used to refer to one of the XML Schema **simple type**, and is left unchanged during normalization.

If SimpleType is an [XML Schema Part 2] **simple type**, then

$$[\text{SimpleType}]_{\text{Datatype}}$$

==

SimpleType

otherwise it is an error.

> **Ed. Note:** Jerome: The formal semantics makes use of several built-in types which are not in XML Schema, notably fs:numeric and fs:UnknownSimpleType. These types are necessary for the specification of some of XPath type conversion rules, and are accepted without raising an error. The status of these types is still an open issue. See [**Issue-0127: Datatype limitations**].

4.13.2 Expressions on Datatypes

Introduction

Expressions on Datatypes are expressions whose semantics depends on the type of some of the sub-expressions on which they are applied. The syntax of Datatype expressions is described by the following grammar productions.

[16]	InstanceofExpr	::=	Expr "instance" "of" "only"? Datatype
[10]	TypeswitchExpr	::=	"typeswitch" "(" Expr ")" ("as" Variable)? CaseClause+ "default" "return" Expr
[30]	CaseClause	::=	"case" Datatype "return" Expr

[23] CastExpr ::= ("cast" "as" | "treat" "as" | "assert" "as") Datatype "(" Expr ")"

4.13.2.1 Instance of

Introduction

The Datatype expression "*Expr* instance of *Datatype*" is true if and only if the result of evaluating expression *Expr* is an instance of the type referred to by *Datatype*.

Normalization

An "instance of" expression is normalized into a "typeswitch" expression. Note that the following normalization rule uses a variable $fs:new, which is a newly created variable which must not conflict with any variables in scope. This variable is necessary to comply to the syntax of typeswitch expressions in the core XQuery, but is never used.

$$\Big[\textit{Expr}\ \text{instance of}\ \textit{Datatype}\Big]_{\text{Expr}} \;==\; \text{typeswitch}\ (\Big[\ \textit{Expr}\ \Big]_{\text{Expr}})\ \text{as}\ \$\text{fs:new}\ \ \text{case}\ \textit{Datatype}\ \text{return xf:true()}\ \ \text{default return xf:false()}$$

> **Ed. Note:** MFF: The "instance of" expression allows an optional "only" modifier. The use case for such a modifier is based on named typing, while the XQuery semantics is currently based on structural typing. It is not clear what the semantics of the "only" modifier under structural typing should be and how it can be supported. See [**Issue-0111: Semantics of instance of ... only**].

4.13.2.2 Cast expressions

Cast expressions are expressions that check or change the type of an expression against a given type.

4.13.2.2.1 Cast as

Introduction

The expression "cast as *Datatype* (*Expr*)" can be used to explicitly convert the result of an expression from one type to another. It changes both the type and value of the result of an expression, and can only be applied on a simple value and an XML Schema simple type.

The semantics of cast expressions follows the specification given in Section [14. Casting Functions] of the [XQuery 1.0 and XPath 2.0 Functions and Operators] document. The *casting table* in Section [14. Casting Functions] of the [XQuery 1.0 and XPath 2.0 Functions and Operators] document indicates whether a cast is allowed or not. In case it is allowed, a specific cast function is applied, based on the input and output XML Schema simple types. The semantics of the cast function follows casting rules which are described in the rest of the remainder of Section [14. Casting Functions] of the [XQuery 1.0 and XPath 2.0 Functions and Operators] document and is not specified further here.

> **Ed. Note:** Jerome: The [XQuery 1.0 and XPath 2.0 Functions and Operators] document does not provide any function for casting, just a table and casting rules. It would be preferable to either

have an explicit function to normalize to, or to put the semantics of casts in the Formal Semantics. This relates to Issue 17 in the [XQuery 1.0 and XPath 2.0 Functions and Operators] document.

Notation

We make use of the following auxiliary judgments to represent access to the casting table and to the semantics of casting, as described in Section [14. Casting Functions] of the [XQuery 1.0 and XPath 2.0 Functions and Operators] document.

The notation:

$$Type_1 \text{ CastAllowed } Type_2 \Rightarrow \{ \text{ Y, M, N } \}$$

indicates whether casting from type $Type_1$ to $Type_2$ is always possible (Y), may be possible (M), or is not allowed (N).

The notation:

$$\text{cast as } Type_2 \text{ (} value_1 \text{) } \Rightarrow value_2$$

indicates that applying the casting rules for $Type_2$ on $value_1$ yields the value $value_2$.

Dynamic Evaluation

If the cast is allowed (Y or M), the following evaluation rule applies the casting rules on the result of the input expression. The rule uses the data model function `dm:type` in order to obtain the dynamic type of the input value, Datatype normalization to obtain the output type, and the above auxiliary judgments to check whether the cast is allowed and apply the casting rules.

$$\frac{\begin{array}{c}\text{dynEnvs} \vdash Expr_1 \Rightarrow value_1 \\ \text{dm:type}(value_1) = Type_1 \\ \left[Datatype_2 \right]_{\text{Datatype}} \Rightarrow Type_2 \\ Type_1 \text{ CastAllowed } Type_2 \Rightarrow \{ \text{ Y, M } \} \\ \text{cast as } Type_2 \text{ (} value_1 \text{) } \Rightarrow value_2\end{array}}{\text{dynEnvs} \vdash \text{cast as } Datatype_2 \text{ (} Expr_1 \text{) } \Rightarrow value_2}$$

Note that in the case the cast is allowed, but the casting table indicates "M", the casting operation might still fail at run-time, if the input value is inappropriate (e.g. attempting to cast the string "VRAI" into xs:boolean). In that case, the dynamic evaluation returns an error value.

In the case the casting table returns "N", the cast is not allowed and the dynamic semantics always returns an error value.

$$\frac{\text{dynEnvs} \text{ |- } Expr_1 \text{ => } value_1 \quad \text{dm:type}(value_1) = Type_1}{\left[\ Datatype_2\ \right]_{\text{Datatype}} \text{ => } Type_2}$$

$$\frac{Type_1 \text{ CastAllowed } Type_2 \text{ => N}}{\text{dynEnvs} \text{ |- cast as } Datatype_2 \text{ (} Expr_1 \text{) => dm:error()}}$$

Static Type Analysis

The following static typing rules gives the statics semantics of "cast as" expression. If the cast table indicates that the cast is allowed, then the static semantics is always the output type of the cast. In the case the cast table indicates the cast is not allowed, the system raises a static type error.

$$\frac{\text{statEnvs} \text{ |- } Expr : Type_1 \qquad \left[\ Datatype_2\ \right]_{\text{Datatype}} \text{ => } Type_2 \qquad Type_1 \text{ CastAllowed } Type_2 \text{ => \{ Y, M \}}}{\text{statEnvs} \text{ |- cast as } Datatype_2 \text{ (} Expr \text{) : } Type_2}$$

4.13.2.2.2 Treat as

Introduction

The expressions "treat as *Datatype* (*Expr*)", can be used to change the dynamic type of the result of an expression, without changing its value. Treat as never raises a static type error, but might raise a run-time error if the dynamic type of the expression is not an instance of the specified type.

Normalization

Treat as expressions are normalized to typeswitch expressions. Note that the following normalization rule uses a variable $fs:new, which is a newly created variable which must not conflict with any variables in scope.

$$\left[\text{treat as } Datatype \text{ (} Expr \text{)}\right]_{\text{Expr}} \quad ==$$

typeswitch ($[$ *Expr* $]_{Expr}$) as $fs:new
case *Datatype* return $fs:new
default return dm:error()

4.13.2.2.3 Assert as

Introduction

The expression "assert as *Datatype* (*Expr*)", performs a similar operation as "treat as" but is always type safe. The system raises a static error in the case it cannot infer that the type of the expression is not a subtype of the specified type.

Dynamic Evaluation

Dynamically, the "assert as" expression is a no-op. As the static type system enforces statically that the type of the input expression is of the specified type.

$$\frac{\text{dynEnvs} \vdash \textit{Expr} \Rightarrow \textit{value}}{\text{dynEnvs} \vdash \text{assert as } \textit{Type} \text{ (} \textit{Expr} \text{)} \Rightarrow \textit{value}}$$

Static Type Analysis

The "assert as" expression is type safe if and only if the type of the input expression is a subtype of the specified type. This semantics is specified as the following typing rule.

$$\frac{\text{statEnvs} \vdash : \textit{Type}_1 \quad [\ \textit{Datatype}_2\ ]_{\text{Datatype}} \Rightarrow \textit{Type}_2 \quad \textit{Type}_1 <: \textit{Type}_2}{\text{statEnvs} \vdash \text{assert as } \textit{Datatype}_2 \text{ (} \textit{Expr}_1 \text{)} : \textit{Type}_2}$$

> **Ed. Note:** Jerome: The semantics for "assert as" here relies on structural typing. It will have to be revised when named typing is added. See **[Issue-0104: Support for named typing]**.

4.13.2.3 Typeswitch

Introduction

The **typeswitch** expression of XQuery allows users to perform different operations according to the type of an input expression.

A typeswitch expression may have an optional "as *Variable*" statement, used to bind a variable to the result of the input expression. This variable is optional in XQuery but mandatory in the XQuery core. As we will see, one of the reasons for having this variable is better static typing.

Normalization

Normalization of typeswitch expressions is applied to make sure an appropriate "as *Variable*" statement is present.

The following general normalization rule merely adds a newly created variable, which does not appear in the rest of the query. In that normalization rule, $Expr_0$ must not be a variable. Note that $fs:new is a newly

generated variable that must not conflict with any variables in scope and is not used in any of the sub-expressions.

$$
\begin{array}{l}
\left[\right. \\
\quad \text{typeswitch (} \mathit{Expr}_0 \text{)} \\
\quad\quad \text{case } \mathit{Datatype}_1 \text{ return } \mathit{Expr}_1 \\
\quad\quad \ldots \\
\quad\quad \text{case } \mathit{Datatype}_n \text{ return } \mathit{Expr}_n \\
\quad\quad \text{default return } \mathit{Expr}_{n+1} \\
\left.\right]_{\text{Expr}}
\end{array}
$$

==

$$
\begin{array}{l}
\text{typeswitch (} \left[\right. \mathit{Expr}_0 \left.\right]_{\text{Expr}} \text{) as \$fs:new} \\
\quad \text{case } \mathit{Datatype}_1 \text{ return } \left[\right. \mathit{Expr}_1 \left.\right]_{\text{Expr}} \\
\quad \ldots \\
\quad \text{case } \mathit{Datatype}_n \text{ return } \left[\right. \mathit{Expr}_n \left.\right]_{\text{Expr}} \\
\quad \text{default return } \left[\right. \mathit{Expr}_{n+1} \left.\right]_{\text{Expr}}
\end{array}
$$

An additional normalization rule is also applied in the case where the input expression is limited to a variable. This special case rule is used to propagate better typing information in each "case" branch of the typeswitch.

$$
\begin{array}{l}
\left[\right. \\
\quad \text{typeswitch (} \mathit{Variable}_0 \text{)} \\
\quad\quad \text{case } \mathit{Datatype}_1 \text{ return } \mathit{Expr}_1 \\
\quad\quad \ldots \\
\quad\quad \text{case } \mathit{Datatype}_n \text{ return } \mathit{Expr}_n \\
\quad\quad \text{default return } \mathit{Expr}_{n+1} \\
\left.\right]_{\text{Expr}}
\end{array}
$$

==

typeswitch ($Variable_0$) as $Variable_0$

case $Datatype_1$ return $\left[Expr_1 \right]_{Expr}$

...

case $Datatype_n$ return $\left[Expr_n \right]_{Expr}$

default return $\left[Expr_{n+1} \right]_{Expr}$

In other words, this normalization rules treats:

`typeswitch (`*`Variable`*`) case ...`

identically to:

`typeswitch (`*`Variable`*`) as `*`Variable`*` case ...`

As a result, *Variable* benefits from the more accurate type information given to the variable in the "as" clause of the typeswitch during static type analysis.

Notation

We use the following additional grammar production to identify branches of the typeswitch.

[101] CaseRules ::= ("case" Datatype "return" Expr CaseRules) | ("default" "return" Expr)

When defining the dynamic and static semantics of typeswitch, we use the following auxiliary judgments to denote partial evaluation and typing for the branches of a typeswitch.

The two following judgments

dynEnvs Case (*Variable*, *value*) |- case *Type* return *Expr CaseRules* => $value_2$

and

dynEnvs Case (*Variable*, *value*) |- default return *Expr* => $value_2$

are used in the dynamic semantics of typeswitch. They indicates that under the environment dynEnvs, and with the typeswitch variable "*Variable*" bound to value "*value*", the given case rules (e.g., "case *Type* return *Expr CaseRules*") evaluate to $value_2$.

The two following judgments

statEnvs Case (*Variable*, *Type* ; ($Type_1$ | ... | $Type_n$)) |- case *Datatype* return *Expr CaseRules* : $Type_2$

and

statEnvs Case (*Variable*, *Type* ; ($Type_1$ | ... | $Type_n$))) |- default return *Expr* : $Type_2$

are used in the static semantics of typeswitch. They indicates that under the type environment statEnvs, and with the typeswitch variables "*Variable*" bound to type "*Type*", the type inferred for the given case rules (e.g., "case *Datatype* return *Expr CaseRules*") is type $Type_2$. Note that the typing judgments also keep track of all previously visited types in the typeswitch. This additional information is used later on in typing the default clause.

Dynamic Evaluation

The evaluation of a typeswitch proceeds as follows. First, the input expression is evaluated, yielding an input value. Then the first case clause whose *Datatype* type matches that value is selected and its corresponding expression is evaluated.

During typeswitch evaluation, the variable *Variable* and the value *value* to match against, are kept on the left of the turnstile |- in the auxiliary judgments. Each case rule of a typeswitch expression is always evaluated *against* this value, and case rules are tried from left to right. The rule for the typeswitch expression evaluates its expression and sets up the appropriate environment for the case rules:

$$\frac{\text{dynEnvs} \vdash \mathit{Expr} \Rightarrow \mathit{value}_0 \qquad \text{dynEnvs Case } (\mathit{Variable}, \mathit{value}_0) \vdash \mathit{CaseRules} \Rightarrow \mathit{value}_1}{\text{dynEnvs} \vdash \text{typeswitch } (\mathit{Expr}) \text{ as } \mathit{Variable}\ \mathit{CaseRules} \Rightarrow \mathit{value}_1}$$

If the value value_0 is in the domain of the type *Type*, the next rule extends the environment by binding the variable *Variable* to value_0 and evaluates the body of the case rule. Remember that the *domain* of a type is the possibly infinite set containing all values that are instances of that type.

> **Ed. Note:** Jerome: The notion of domain of a type needs to be clarified. This should go in section 3. about the type system. The notion of domain of a type will change as soon as the named typing proposal is added. See **[Issue-0104: Support for named typing]**.

$$\frac{\left[\ \mathit{Datatype}\ \right]_{\text{Datatype}} \Rightarrow \mathit{Type} \qquad \mathit{value}_0 \texttt{ in } \text{Dom}(\mathit{Type}) \qquad \text{dynEnvs } [\ \text{varValue}(\mathit{Variable} \mid\!\!\rightarrow \mathit{value})\ ] \vdash \mathit{Expr} \Rightarrow \mathit{value}_1}{\text{dynEnvs Case } (\mathit{Variable}, \mathit{value}_0) \vdash \text{case } \mathit{Datatype} \text{ return } \mathit{Expr}\ \mathit{CaseRules} \Rightarrow \mathit{value}_1}$$

If the value value_0 is not in the domain of the type expression *Type* (i.e., the normalized version of *Datatype*),the next rule evaluates the case rules following the current one. The body of the given case rule is not evaluated if value_0 is not in the domain of the given type.

$$\frac{\left[\ \mathit{Datatype}\ \right]_{\text{Datatype}} \Rightarrow \mathit{Type} \qquad \text{not } (\mathit{value}_0 \texttt{ in } \text{Dom}(\mathit{Type})) \qquad \text{dynEnvs Case } (\mathit{Variable}, \mathit{value}_0) \vdash \mathit{CaseRules} \Rightarrow \mathit{value}_1}{\text{dynEnvs Case } (\mathit{Variable}, \mathit{value}_0) \vdash \text{case } \mathit{Datatype} \text{ return } \mathit{Expr}\ \mathit{CaseRules} \Rightarrow \mathit{value}_1}$$

Finally, the last rule states that the "default" branch of a typeswitch expression always evaluates to its given expression.

$$\frac{\text{dynEnvs} \vdash \mathit{Expr} \Rightarrow \mathit{value}_1}{\text{dynEnvs Case } (\mathit{Variable}, \mathit{value}_0); \vdash \text{default return } \mathit{Expr} \Rightarrow \mathit{value}_1}$$

Static Type Analysis

The typeswitch expression possesses one of the more complex sets of static typing rules. The rules account for the fact that if the static type of the conditional expression is known, then we may be able to determine that some of the case clauses do not apply.

The main typeswitch rule relies upon the auxiliary type judgments to determine the type of each of the case clauses and of the default clause. These rules are provided after the main rule. Note the type of the input expression is always treated as a collection of similar items, using the "prime" and "quantifier" operations on types. This is necessary as further typing rules compute the common prime types for branch of the type switch.

Ed. Note: Jerome: the use of the common prime types replaces the previous use of type intersection. Common prime types simplifies significantly the complexity in implementing typeswitch, but is less precise in certain cases.

$$\frac{\begin{array}{c}\text{statEnvs} \vdash Expr_0 : Type_0 \\ Type_0' = \text{prime}(Type_0) \cdot \text{quantifier}(Type_0) \\ \left[\ Datatype_1\ \right]_{\text{Datatype}} \Rightarrow Type_1 \\ \cdots \\ \left[\ Datatype_n\ \right]_{\text{Datatype}} \Rightarrow Type_n \\ \text{statEnvs Case } (Variable_0, Type_0\ ;\ ()) \vdash \text{case } Datatype_1 \text{ return } Expr_1 : Type_1' \\ \cdots \\ \text{statEnvs Case } (Variable_0, Type_0\ ;\ (Type_1 \mid \ldots \mid Type_{n-1})) \vdash \text{case } Datatype_n \text{ return } Expr_n : Type_n' \\ \text{statEnvs Case } (Variable_0, Type_0\ ;\ (Type_1 \mid \ldots \mid Type_n)) \vdash \text{default return } Expr_{n+1} : Type_{n+1}'\end{array}}{\begin{array}{l}\text{statEnvs} \vdash (\text{typeswitch } (Expr_0) \text{ as } Variable_0 \\ \quad \text{case } Datatype_1 \text{ return } Expr_1 \\ \quad \cdots \\ \quad \text{case } Datatype_n \text{ return } Expr_n \\ \quad \text{default return } Expr_{n+1}) \qquad : Type_1' \mid \ldots \mid Type_{n+1}'\end{array}}$$

Now we give the rules that determines the static type of each case clause in the typeswitch. In each rule, one need to compute the "common prime types" between the input type and the case clause datatype.

The first rule is applied if the "common prime types" is none. In that case, we know for sure the corresponding case clause will not be evaluated and the corresponding result type is none. Thanks to this rule, it is often possible to infer a quite precise type for the overall typeswitch through elimination of some branches.

$$\frac{\begin{array}{c}\left[\ Datatype\ \right]_{\text{Datatype}} \Rightarrow Type \\ \text{common-primes}(\text{prime}(Type_0), \text{prime}(Type)) = Type'_0 \\ Type'_0 = \text{none}\end{array}}{\text{statEnvs Case } (Variable_0, Type_0\ ;\ (Type_1 \mid \ldots \mid Type_r)) \vdash \text{case } Datatype \text{ return } Expr : \text{none}}$$

The second rule is applied if the "common prime types" is anything else but none. In that case, the input variable is added into the type environment, and type inference is applied on the expression on the right-hand side of the case clause. Note that the type of the input variable is set to the "common prime types", and not the input type.

$$\frac{\begin{array}{c}\left[\ Datatype\ \right]_{\text{Datatype}} \Rightarrow Type \\ \text{common-primes}(\text{prime}(Type_0), \text{prime}(Type)) = Type'_0 \\ \text{common-occur}(\text{quantifier}(Type_0), \text{quantifier}(Type)) = Quant_0' \\ \text{not}(Type'_0 = \text{none}) \\ \text{statEnvs}\ [\ \text{varType}(Variable : Type'_0 \cdot Quant_0')\ ]\ \text{|-}\ Expr : Type_1\end{array}}{\text{statEnvs}\ \text{|-}\ \text{case}\ Datatype\ \text{return}\ Expr : Type_1}$$

Note that these two rules do not take the visited datatypes into account. The "default" clause differs from the other clauses is that it does not specify a Datatype. The typing rule for the "default" clause uses the visited type instead. Intuitively, the type corresponding to the "default" clause is *any type but the ones in the other cases clauses*.

Therefore, in case the type of the input expression is a subtype of all the visited types, then one knows for sure the case clause is not evaluated and the type of the default clause is xq_none;.

$$\frac{Type_0 <: (Type_1 \mid ... \mid Type_n)}{\text{statEnvs}\ \text{Case}\ (Variable_0, Type_0\ ;\ (Type_1 \mid ... \mid Type_n))\ \text{|-}\ \text{default return}\ Expr : \text{none}}$$

Otherwise, the input variable is added into the type environment, and type inference is applied on the expression on the right-hand side of the default clause. Note that the type of the input variable is set to the input type of the expression.

Ed. Note: Jerome: There is an asymmetry here. It would be nicer to be able to have the type be more precise, like for the other case clauses. The technical problem is the need for some form of negation. I think one could define a "non-common-primes" function that would do the trick, but I leave that as open for now until further review of the new typeswitch section is made. See [**Issue-0112: Typing for the typeswitch default clause**].

$$\frac{\begin{array}{c}\text{not}(Type_0 <: (Type_1 \mid ... \mid Type_n)) \\ \text{statEnvs}\ [\ \text{varType}(Variable : Type_0)\ ]\ \text{|-}\ Expr : Type_1\end{array}}{\text{statEnvs}\ \text{Case}\ (Variable_0, Type_0\ ;\ (Type_1 \mid ... \mid Type_n))\ \text{|-}\ \text{default return}\ Expr : Type_1}$$

Example

The typing rules for typeswitch provides reasonably precise type information in a number of useful cases. For example, consider the following a iteration expression followed by a typeswitch.

```
for $x in $bib/book/*
return
   typeswitch $x as $e
   case element author return $e/name
   case element title return &fs_data;($e)
   default return ()
```

Remember that the "data" function is only working on single nodes. Note that the above typeswitch is using

variable `$e`, which has a more precise typing (`element title { xs:string }`) than the input expression `$x` (`element title | element author | element editor ...`).

The type rules for typeswitch do not, however, account for the interdependence between successive case clauses. Thus if two case clauses had overlapping *Datatype*s, the static rules would behave as if both case clauses "fired", rather than just the first one.

> **Ed. Note:** Jerome: It seems that the simpler version of typeswitch proposed here would actually allow to take previous case clauses into account. This is something worth exploring as it would improve the static analysis in a way that might be helpful to users. See [**Issue-0112: Typing for the typeswitch default clause**].

5 The Query Prolog

> **Ed. Note: Status:** This section is still draft and needs further revision. Revision of the section is pending agreement on issues related to namespaces and named typing.

Introduction

This section defines the semantics of [3. The Query Prolog] in [XQuery 1.0: A Query Language for XML].

The **Query Prolog** is a series of declarations and definitions that affect query processing. The Query Prolog can be used to define namespaces, import type definitions from XML Schemas, and define functions. Namespace declarations and schema imports always precede function definitions, as specified by the following grammar productions.

[1]	Query	::=	QueryProlog ExprSequence?
[2]	QueryProlog	::=	(NamespaceDecl | DefaultNamespaceDecl | SchemaImport)* FunctionDefn*

The order in which functions are defined is immaterial. Notably, user-defined functions may invoke other user-defined functions in any order.

5.1 Namespace Declarations and Schema Imports

Introduction

[68]	NamespaceDecl	::=	"namespace" QName "=" StringLiteral
[69]	DefaultNamespaceDecl	::=	"default" ("element" | "function") "namespace" "=" StringLiteral
[72]	SchemaImport	::=	"schema" StringLiteral ("at" StringLiteral)?

Namespace Declarations and Schema Import are not part of query proper but are used to modify the input context for the rest of the query processing. As such, Namespace Declarations and Schema Import are processed before the normalization phase.

The semantics of Schema Import is described in terms of the XQuery type system. The process of converting an XML Schema into a sequence of type declarations is described in Section **[3.5 Importing types from XML Schema]**. We here describe how the resulting sequence of type declarations is added into the static environment when the prolog is processed.

Notation

We denote by prolog declarations, either namespace declarations or type declarations.

[73]	PrologDeclList	::=	PrologDecl*
[74]	PrologDecl	::=	NamespaceDecl
			| DefaultNamespaceDecl
			| TypeDeclaration

When processing Namespace Declarations and Schema Import, we use the following auxiliary judgment. The notation:

PrologDeclList => statEnvs

indicates that the sequence of prolog declarations *PrologDeclList* yields the static environment statEnvs.

Context Processing

Prolog declarations are processed in the order they are encountered, as described by the following inference rules. The first rule specify that for an empty sequence of prolog declarations, the static environment is composed of a default context.

> **Ed. Note:** Jerome: What do the default namespace and type environments contain? I believe at least the default namespace environment should contain the "xs", "xf" and "op" prefixes, as well as the default namespaces bound to the empty namespace. Should the default type environment contain wildcard types? See **[Issue-0115: What is in the default context?]**.

$$\frac{}{() \Rightarrow \text{statEnvsDefault}}$$

A namespace declaration adds a new (prefix,uri) binding in the namespace component of the static environment.

$$\frac{\textit{PrologDeclList} \Rightarrow \text{statEnvs} \quad \text{statEnvs}' = \text{statEnvs} [\ \text{namespace}(\textit{NCName} \mid\!\!\rightarrow \textit{StringLiteral})\ ]}{\textit{PrologDeclList}\ \text{namespace}\ \textit{NCName} = \textit{StringLiteral} \Rightarrow \text{statEnvs}'}$$

A default element namespace declaration changes the default element namespace prefix binding in the namespace component of the static environment.

$$\frac{\textit{PrologDeclList} \Rightarrow \text{statEnvs} \quad \text{statEnvs}' = \text{statEnvs} [\ \text{namespace}(\text{fsdefaultelem} \mid\!\!\rightarrow \textit{StringLiteral})\ ]}{\textit{PrologDeclList}\ \text{default element namespace} = \textit{StringLiteral} \Rightarrow \text{statEnvs}'}$$

A default function namespace declaration changes the default function namespace prefix binding in the namespace component of the static environment.

$$\frac{\textit{PrologDeclList} \Rightarrow \text{statEnvs} \quad \text{statEnvs}' = \text{statEnvs} [\ \text{namespace}(\text{fsdefaultfunc} \mid\!\!\rightarrow \textit{StringLiteral})\ ]}{\textit{PrologDeclList}\ \text{default function namespace} = \textit{StringLiteral} \Rightarrow \text{statEnvs}'}$$

Type, element and attribute declarations are added respectively to the type, element and attribute declarations components of the static environment.

PrologDeclList => statEnvs
statEnvs' = statEnvs [typeDecl(*QName* |-> *Type*)]

PrologDeclList define type *QName* { *Type* } => statEnvs'

PrologDeclList => statEnvs
statEnvs' = statEnvs [elemDecl(*QName* |-> *Type*)]

PrologDeclList define element *QName* { *Type* } => statEnvs'

PrologDeclList => statEnvs
statEnvs' = statEnvs [attrDecl(*QName* |-> *Type*)]

PrologDeclList define attribute *QName* { *Type* } => statEnvs'

Note that for namespaces, later declarations can override earlier declarations of the same prefix. In the case of global elements, attributes and types, multiple declarations correspond to an error.

5.2 Function Definitions

Introduction

User defined functions specify the name of the function, the names and types of the parameters, and the type of the result. The **function body** defines how the result of the function is computed from its parameters.

[70]	FunctionDefn	::=	"define" "function" QName "(" ParamList? ")" ("returns" Datatype)? EnclosedExpr
[71]	ParamList	::=	Param ("," Param)*
[52]	Param	::=	Datatype? Variable

Notation

We use the following auxiliary normalization rule $[\cdot\cdot\cdot]_{\text{Param}}$ for the normalization of parameters in function definitions.

PrologDeclList => statEnvs
statEnvs' = statEnvs [attrDecl(*QName* |-> *Type*)]

PrologDeclList define attribute *QName* { *Type* } => statEnvs'

Normalization

The only form of normalization required for user defined functions is adding the type for its parameters or for the return clause if it is not provided.

$$\left[\text{ define function } \mathit{QName} \text{ (} \mathit{ParamList}? \text{) returns } \mathit{Datatype}\ \mathit{EnclosedExpr} \right]_{\text{Expr}}$$
$$==$$
$$\text{define function } \left[\mathit{QName}\right] \text{ (} \left[\mathit{ParamList}?\right]_{\text{Param}} \text{) returns } \mathit{Datatype} \left[\mathit{EnclosedExpr}\right]_{\text{Expr}}$$

If the return type of the function is not provided, it is given the item* datatype, corresponding to xs:AnyType.

$$\left[\text{ define function } \mathit{QName} \text{ (} \mathit{ParamList}? \text{) } \mathit{EnclosedExpr} \right]_{\text{Expr}}$$
$$==$$
$$\text{define function } \left[\mathit{QName}\right] \text{ (} \left[\mathit{ParamList}?\right]_{\text{Param}} \text{) returns item* } \left[\mathit{EnclosedExpr}\right]_{\text{Expr}}$$

Parameters without a declared typed are given the item* datatype, corresponding to xs:AnyType.

$$\left[\mathit{Variable}\right]_{\text{Param}}$$
$$==$$
$$\text{item* } \mathit{Variable}$$

$$\left[\mathit{Datatype}\ \mathit{Variable}\right]_{\text{Param}}$$
$$==$$
$$\mathit{Datatype}\ \mathit{Variable}$$

Context Processing

First, all the function signatures are added into the static environment, and all the function bodies are added into the dynamic environment. This process happens before static type analysis occurs.

$$\frac{\begin{array}{c}\mathit{FunctionDefnList} \Rightarrow \text{statEnvs}, \text{dynEnvs} \\ \text{statEnvs}' = \text{statEnvs} \left[\text{funcType}(\mathit{QName} \text{ |-> (} \mathit{Datatype}_1 \text{ , } \cdots \mathit{Datatype}_n \text{ , } \mathit{Datatype}_r)) \right] \\ \text{dynEnvs}' = \text{dynEnvs} \left[\text{funcDefn}(\mathit{QName} \text{ |-> (} \mathit{Expr} \text{ , } \mathit{Variable}_1 \text{ , } \cdots \mathit{Variable}_n)) \right]\end{array}}{\begin{array}{l}\mathit{FunctionDefnList} \text{ define function } \mathit{QName} \text{ (} \mathit{Datatype}_1\ \mathit{Variable}_1, \cdots \mathit{Datatype}_n\ \mathit{Variable}_n \text{)} \\ \text{returns } \mathit{Datatype}_r \\ \{ \mathit{Expr} \} \Rightarrow \text{statEnvs}', \text{dynEnvs}'\end{array}}$$

Static Type Analysis

The static typing rules for function definitions checks whether the type of the enclosed expression is consistent with the type of the input parameters, and the type of the return clause.

$$\frac{\begin{array}{c}\text{statEnvs} \left[\text{varType}(\mathit{Variable}_1 : \mathit{Datatype}_1 \text{ ;...; } \mathit{Variable}_n : \mathit{Datatype}_n \text{) } \right] \text{ |- } \mathit{Expr} : \mathit{Type} \\ \mathit{Type} <: \mathit{Datatype}_r\end{array}}{\begin{array}{l}\text{statEnvs} \text{ |- define function } \mathit{QName} \text{ (} \mathit{Datatype}_1\ \mathit{Variable}_1 \text{ , } \cdots \mathit{Datatype}_n\ \mathit{Variable}_n \text{)} \\ \text{returns } \mathit{Datatype}_r \\ \{ \mathit{Expr} \}\end{array}}$$

What this typing rule is checking is: if the input parameters are of the given type, then is it true that the result of the function is of the return type. If the type checking fails, the system raises an error. Otherwise, it does not have any other effect, as function signatures are already inside the static environment.

Dynamic Evaluation

There is no need to describe a dynamic semantics at this point, as we do not have yet any actual value to evaluated the function. The actual semantics of function evaluation has been described in **[4.2.4 Function Calls]**.

6 Additional Semantics of Functions

> **Ed. Note: Status:** This section is still incomplete. This section will be completed as soon as Sections 4 and 5 are consolidated. See [**Issue-0135: Semantics of special functions**].

As was explained in section **[2.4 Functions]**, a number of functions play a role defining the formal semantics of XQuery. Some other functions from the [XQuery 1.0 and XPath 2.0 Functions and Operators] document need special static typing rules. This section gives the semantics of all "special" functions, used in the formal semantics.

6.1 Formal Semantics Functions

Introduction

We give the definition, and semantics, of functions used in the formal semantics that are not part of the [XQuery 1.0 and XPath 2.0 Functions and Operators] document.

6.1.1 The `fs:document` function

The fs:document function computes the document node of the current node, and is used to define the semantics of the `/` expression in XQuery. The fs:document function is defined as a recursive XQuery function as follows.

```
define function fs:document(node $x) returns node
  {
     if empty(dm:parent($x))
     then $x
     else fs:document($x)
  }
```

6.1.2 The `fs:data` function

Introduction

The fs:data function is used to access the value content of an element or attribute. This function corresponds to the dm:typed-value accessor in the XQuery data model.

> **Ed. Note:** Some aspects of the semantics of the data() function are still an open issue. For instance, what should be the result of data() over a text node. See [**Issue-0107: Semantics of data()**].

Notation

To compute the resulting type for the fs:data function, we use the same approach as for the child axis in XPath, by applying the fs:data function as a Filter. See **[4.3.1 Axis Steps]** for the definition and the semantics of Filters.

We use the following notation, adapted from the Filter judgment in **[4.3.1 Axis Steps]**.

dynEnvs.varValue ; $Type_1$ |- fs:data : $Type_2$

Static Type Analysis

Static type analysis for the fs:data function checks that the function is applied on an element or attribute with a simple type content. If so, it returns the corresponding simple type through an application of the Filter judgment on the input type of the function.

$$\frac{\begin{array}{c}\text{statEnvs} \text{ |- } Expr : Type_1 \\ \text{statEnvs} \text{ |- } Type_1 \texttt{ <: } (\text{element } * \{ \text{xs:AnyAttribute}*, \text{xs:AnySimpleType} \} \mid \text{attribute } * \{ \text{xs:AnySimpleType} \}) \\ \text{dynEnvs.varValue} ; Type_1 \text{ |- fs:data} : Type_2\end{array}}{\text{statEnvs} \text{ |- fs:data}(Expr) : Type_2}$$

$$\frac{}{\text{statEnvs} ; \text{element } qname \{ \text{AttrType}, \text{SimpleType} \} \text{ |- fs:data} : \text{SimpleType};}$$

$$\frac{}{\text{statEnvs} ; \text{attribute } qname \{ \text{SimpleType} \} \text{ |- fs:data} : \text{SimpleType};}$$

If applied on any other kind of item, it returns the empty sequence.

$$\frac{\begin{array}{c}\text{statEnvs} \text{ |- } Expr : Type \\ \text{statEnvs} \text{ |- not}(Type \texttt{ <: } (\text{element } * \{ \text{xs:AnyAttribute}*, \text{xs:AnySimpleType} \} \mid \text{attribute } * \{ \\ \text{xs:AnySimpleType} \})) \\ \text{statEnvs} \text{ |- } Type \texttt{ <: item}\end{array}}{\text{dynEnvs} \text{ |- fs:data}(Expr) : ()}$$

Otherwise (for empty sequences or sequences of more than one item value), it raises an error.

Example

Consider the following variables and its corresponding static type.

```
$x : (element price { attribute currency { xs:string }, xs:decimal }
     | element price_code { xs:integer })
```

Applying the fs:data function on that variable results in the following type.

```
fs:data($x) : (xs:decimal | xs:integer)
```

Remark that, as the input type is a choice, applying the Filter judgment results in a choice of simple types for

the output of the fs:data function.

Dynamic Evaluation

Dynamically, the fs:data function is implemented as the dm:typed-value data model accessor.

$$\frac{\text{dynEnvs} \text{ |- } Expr \Rightarrow value_1 \qquad \text{dm:typed-value}(value_1) = value_2}{\text{dynEnvs} \text{ |- fs:data}(Expr) \Rightarrow value_2}$$

6.2 Functions with specific typing rules

6.2.1 The `dm:error` function

Static Type Analysis

The dm:error function always returns the none type.

$$\frac{}{\text{statEnvs} \text{ |- dm:error() : none}}$$

6.2.2 The `op:union`, `op:intersect` and `op:expect` operators

Static Type Analysis

The static semantics for op:union uses the auxiliary type functions prime(*Type*) and quantifier(*Type*); which are defined in **[3.4 Prime types]**. The type of each argument is determined, and then prime(.) and quantifier(.) are applied to the sequence type ($Type_1$, $Type_2$).

$$\frac{\text{statEnvs} \text{ |- } Expr_1 : Type_1 \qquad \text{statEnvs} \text{ |- } Expr_2 : Type_2 \qquad Type_3 = Type_1 \text{ , } Type_2}{\text{statEnvs} \text{ |- op:union}(Expr_1, Expr_2) : \text{prime}(Type_3) \cdot \text{quantifier}(Type_3)}$$

The static semantics of op:intersect is analogous to that for op:union Because an intersection may always be empty, the result type needs to be made optional.

$$\frac{\text{statEnvs} \text{ |- } Expr_1 : Type_1 \qquad \text{statEnvs} \text{ |- } Expr_2 : Type_2 \qquad Type_3 = Type_1, Type_2}{\text{statEnvs} \text{ |- op:intersect}(Expr_1, Expr_2) : \text{prime}(Type_3) \cdot \text{quantifier}(Type_3) \cdot ?}$$

The static semantics of op:except follows. The type of the second argument is ignored as it does not contribute to the result type. Like with op:intersect the result of may be the empty list.

$$\frac{\text{statEnvs} \text{ |- } Expr_1 : Type_1}{\text{statEnvs} \text{ |- op:except}(Expr_1, Expr_2) : \text{prime}(Type_1) \cdot \text{quantifier}(Type_1) \cdot ?}$$

6.2.3 The `op:to` operator

The static semantics of the op:to function states that it always returns an integer sequence:

$$\frac{\text{statEnvs} \vdash Expr_1 : \text{xs:integer} \quad \text{statEnvs} \vdash Expr_2 : \text{xs:integer}}{\text{statEnvs} \vdash \text{op:to}(Expr_1, Expr_2) : \text{xs:integer*}}$$

Ed. Note: MFF: the binary operator "to" is not defined on empty sequences. The [XQuery 1.0 and XPath 2.0 Functions and Operators] document says operands are decimals, while the XQuery document says they are integers. What happens when Expr1 > Expr2? See [**Issue-0119: Semantics of op:to**].

A Normalized core grammar

This section contains the grammar of XQuery after it has been normalized, sometimes referred to as the "core" syntax. The XQuery core is [should be] a minimal, complete, and proper subset of the entire XQuery language. It is minimal, because no normalized expression can be removed without reducing the expressiveness of the language. It is complete, because every expression in the XQuery language can be re-expressed as an expression in the normalized language.

Ed. Note: DD 27/8/2001: Do we really believe it is minimal?

The core syntax is currently not a proper subset of the XQuery syntax defined in [XQuery 1.0: A Query Language for XML]. The following XQuery issues address alignment of the core with the complete syntax: '11: Alternative syntax for element construction (xquery-element-construction)'; and '32: Correspondence of Types (xquery-type-correspondence)' [XQuery 1.0: A Query Language for XML].

The following new issues are related to normalization of XQuery: [**Issue-0007: References: IDREFS, Keyrefs, Joins**], [**Issue-0008: Fixed point operator or recursive functions**], [**Issue-0009: Externally defined functions**], [**Issue-0010: Construct values by copy**], [**Issue-0011: XPath tumbler syntax instead of index?**].

NON-TERMINALS

[1]	Query	::=	QueryProlog ExprSequence?
[2]	QueryProlog	::=	(NamespaceDecl | DefaultNamespaceDecl | SchemaImport)* FunctionDefn*
[3]	ExprSequence	::=	Expr (Comma Expr)*
[4]	Expr	::=	SortExpr | ForExpr | LetExpr | TypeswitchExpr | IfExpr | CastExpr | Constructor | PathExpr

[5]	SortExpr	::=	Expr Stable? Sortby Lpar SortSpecList Rpar
[6]	ForExpr	::=	ForClause Return Expr
[7]	LetExpr	::=	LetClause Return Expr
[8]	TypeswitchExpr	::=	Typeswitch Lpar Expr Rpar As Variable CaseClause+ Default Return Expr
[9]	IfExpr	::=	If Lpar Expr Rpar Then Expr Else Expr
[10]	CastExpr	::=	(CastAs | AssertAs) Datatype Lpar Expr Rpar
[11]	Constructor	::=	ComputedElementConstructor | ComputedAttributeConstructor
[12]	PathExpr	::=	RelativePathExpr
[13]	SortSpecList	::=	Expr (Ascending | Descending) (Comma SortSpecList)?
[14]	ForClause	::=	For Variable In Expr
[15]	LetClause	::=	Let Variable ColonEquals Expr
[16]	CaseClause	::=	Case Datatype Return Expr
[17]	RelativePathExpr	::=	StepExpr
[18]	StepExpr	::=	AxisStep | GeneralStep
[19]	AxisStep	::=	Axis NodeTest
[20]	Axis	::=	AxisChild | AxisDescendant | AxisParent | AxisAttribute | AxisSelf | AxisDescendantOrSelf
[21]	NodeTest	::=	NameTest | KindTest
[22]	NameTest	::=	QName | Wildcard
[23]	Wildcard	::=	Star | NCNameColonStar | StarColonNCName
[24]	KindTest	::=	ProcessingInstructionTest | CommentTest | TextTest | AnyKindTest
[25]	ProcessingInstructionTest	::=	ProcessingInstruction Lpar StringLiteral? Rpar
[26]	CommentTest	::=	Comment Lpar Rpar
[27]	TextTest	::=	Text Lpar Rpar
[28]	AnyKindTest	::=	Node Lpar Rpar
[29]	GeneralStep	::=	PrimaryExpr
[30]	PrimaryExpr	::=	Variable | Literal | ElementNameOrFunctionCall | ParenthesizedExpr
[31]	NumericLiteral	::=	IntegerLiteral | DecimalLiteral | DoubleLiteral
[32]	Literal	::=	NumericLiteral | StringLiteral
[33]	ParenthesizedExpr	::=	Lpar ExprSequence? Rpar

[34]	ElementNameOrFunctionCall	::=	QName Lpar (Expr (Comma Expr)*)? Rpar
[35]	Param	::=	Datatype Variable
[36]	Datatype	::=	((ElementOfType QName) | DTKind | Node | SimpleType | Item) OccurrenceIndicator
[37]	DTKind	::=	(Element | Attribute) QName?
[38]	SimpleType	::=	QName
[39]	OccurrenceIndicator	::=	(Star | Plus | QMark)?
[40]	ComputedElementConstructor	::=	Element (QName | EnclosedExpr) Lbrace ExprSequence? Rbrace
[41]	ComputedAttributeConstructor	::=	Attribute (QName | EnclosedExpr) Lbrace ExprSequence? Rbrace
[42]	EnclosedExpr	::=	Lbrace ExprSequence Rbrace
[43]	NamespaceDecl	::=	Namespace QName Equals StringLiteral
[44]	DefaultNamespaceDecl	::=	Default (Element | Function) Namespace Equals StringLiteral
[45]	FunctionDefn	::=	DefineFunction QName Lpar ParamList? Rpar Returns Datatype EnclosedExpr
[46]	ParamList	::=	Param (Comma Param)*
[47]	SchemaImport	::=	Schema StringLiteral (AtKeyword StringLiteral)?

Normalization also simplifies the lexical structure of the language:

TERMINALS

[50]	S	::=	WhitespaceChar+
[51]	AxisChild	::=	"child" "::"
[52]	AxisDescendant	::=	"descendant" "::"
[53]	AxisParent	::=	"parent" "::"
[54]	AxisAttribute	::=	"attribute" "::"
[55]	AxisSelf	::=	"self" "::"
[56]	AxisDescendantOrSelf	::=	"descendant-or-self" "::"
[57]	AxisNamespace	::=	"namespace" "::"
[58]	DefineFunction	::=	"define" "function"
[59]	AtKeyword	::=	"at"
[60]	In	::=	"in"
[61]	Return	::=	"return"
[62]	Then	::=	"then"
[63]	Else	::=	"else"
[64]	Default	::=	"default"
[65]	Namespace	::=	"namespace"
[66]	As	::=	"as"
[67]	Case	::=	"case"
[68]	Only	::=	"only"
[69]	Returns	::=	"returns"

[70]	Function	::=	"function"
[71]	Element	::=	"element"
[72]	Item	::=	"item"
[73]	Attribute	::=	"attribute"
[74]	ElementOfType	::=	"element" "of" "type"
[75]	TypeToken	::=	"type"
[76]	Node	::=	"node"
[77]	Schema	::=	"schema"
[78]	Nmstart	::=	Letter | "_"
[79]	Nmchar	::=	Letter | CombiningChar | Extender | Digit | "." | "-" | "_"
[80]	Star	::=	"*"
[81]	ColonStar	::=	":" "*"
[82]	NCNameColonStar	::=	":"? NCName ":" "*"
[83]	StarColonNCName	::=	"*" ":" NCName
[84]	Equals	::=	"="
[85]	ColonEquals	::=	":="
[86]	Plus	::=	"+"
[87]	QMark	::=	"?"
[88]	Arrow	::=	"=>"
[89]	Lpar	::=	"("
[90]	Rpar	::=	")"
[91]	Variable	::=	"$" QName
[92]	For	::=	"for"
[93]	Let	::=	"let"
[94]	CastAs	::=	"cast" "as"
[95]	AssertAs	::=	"assert" "as"
[96]	Digits	::=	[0-9]+
[97]	IntegerLiteral	::=	Digits
[98]	DecimalLiteral	::=	("." Digits) | (Digits "." [0-9]*)
[99]	DoubleLiteral	::=	(("." Digits) | (Digits ("." [0-9]*)?)) ([e] | [E]) ([+] | [-])? Digits
[100]	Comment	::=	"comment"
[101]	Text	::=	"text"
[102]	ProcessingInstruction	::=	"processing-instruction"
[103]	If	::=	"if"
[104]	Typeswitch	::=	"typeswitch"
[105]	Comma	::=	","
[106]	StringLiteral	::=	(["] [^"]* ["]) | (['] [^']* ['])
[107]	Sortby	::=	"sortby"
[108]	Stable	::=	"stable"
[109]	Ascending	::=	"ascending"
[110]	Descending	::=	"descending"
[111]	PITarget	::=	NCName
[112]	NCName	::=	Nmstart Nmchar*
[113]	QName	::=	":"? NCName (":" NCName)?

[114]	Amp	::=	"&"
[115]	PredefinedEntityRef	::=	"&" ("lt" | "gt" | "amp" | "quot" | "apos") ";"
[116]	HexDigits	::=	([0-9] | [a-f] | [A-F])+
[117]	CharRef	::=	"&#" (Digits | ("x" HexDigits)) ";"
[118]	Lbrace	::=	"{"
[119]	Rbrace	::=	"}"
[120]	LCurlyBraceEscape	::=	"{{"
[121]	RCurlyBraceEscape	::=	"}}"
[122]	Char	::=	([#x0009] | [#x000D] | [#x000A] | [#x0020-#xFFFD])
[123]	WhitespaceChar	::=	([#x0009] | [#x000D] | [#x000A] | [#x0020])
[124]	Whitespace	::=	WhitespaceChar*
[125]	Letter	::=	BaseChar | Ideographic
[126]	BaseChar	::=	([#x0041-#x005A] | [#x0061-#x007A] | [#x00C0-#x00D6] | [#x00D8-#x00F6] | [#x00F8-#x00FF] | [#x0100-#x0131] | [#x0134-#x013E] | [#x0141-#x0148] | [#x014A-#x017E] | [#x0180-#x01C3] | [#x01CD-#x01F0] | [#x01F4-#x01F5] | [#x01FA-#x0217] | [#x0250-#x02A8] | [#x02BB-#x02C1] | [#x0386] | [#x0388-#x038A] | [#x038C] | [#x038E-#x03A1] | [#x03A3-#x03CE] | [#x03D0-#x03D6] | [#x03DA] | [#x03DC] | [#x03DE] | [#x03E0] | [#x03E2-#x03F3] | [#x0401-#x040C] | [#x040E-#x044F] | [#x0451-#x045C] | [#x045E-#x0481] | [#x0490-#x04C4] | [#x04C7-#x04C8] | [#x04CB-#x04CC] | [#x04D0-#x04EB] | [#x04EE-#x04F5] | [#x04F8-#x04F9] | [#x0531-#x0556] | [#x0559] | [#x0561-#x0586] | [#x05D0-#x05EA] | [#x05F0-#x05F2] | [#x0621-#x063A] | [#x0641-#x064A] | [#x0671-#x06B7] | [#x06BA-#x06BE] | [#x06C0-#x06CE] | [#x06D0-#x06D3] | [#x06D5] | [#x06E5-#x06E6] | [#x0905-#x0939] | [#x093D] | [#x0958-#x0961] | [#x0985-#x098C] | [#x098F-#x0990] | [#x0993-#x09A8] | [#x09AA-#x09B0] | [#x09B2] | [#x09B6-#x09B9] | [#x09DC-#x09DD] | [#x09DF-#x09E1] | [#x09F0-#x09F1] | [#x0A05-#x0A0A] | [#x0A0F-#x0A10] | [#x0A13-#x0A28] | [#x0A2A-#x0A30] | [#x0A32-#x0A33] | [#x0A35-#x0A36] | [#x0A38-#x0A39] | [#x0A59-#x0A5C] | [#x0A5E] | [#x0A72-#x0A74] | [#x0A85-#x0A8B] | [#x0A8D] | [#x0A8F-#x0A91] | [#x0A93-#x0AA8] | [#x0AAA-#x0AB0] | [#x0AB2-#x0AB3] | [#x0AB5-#x0AB9] | [#x0ABD] | [#x0AE0] | [#x0B05-#x0B0C] | [#x0B0F-#x0B10] | [#x0B13-#x0B28] | [#x0B2A-#x0B30] | [#x0B32-#x0B33] | [#x0B36-#x0B39] | [#x0B3D] | [#x0B5C-#x0B5D] | [#x0B5F-#x0B61] | [#x0B85-#x0B8A] | [#x0B8E-#x0B90] | [#x0B92-#x0B95] | [#x0B99-#x0B9A] | [#x0B9C] | [#x0B9E-#x0B9F] | [#x0BA3-#x0BA4] | [#x0BA8-#x0BAA] | [#x0BAE-#x0BB5] | [#x0BB7-#x0BB9] | [#x0C05-#x0C0C] | [#x0C0E-#x0C10] | [#x0C12-#x0C28] | [#x0C2A-#x0C33] | [#x0C35-#x0C39] | [#x0C60-#x0C61] | [#x0C85-#x0C8C] | [#x0C8E-#x0C90] | [#x0C92-#x0CA8] | [#x0CAA-#x0CB3] | [#x0CB5-#x0CB9] | [#x0CDE] | [#x0CE0-#x0CE1] | [#x0D05-#x0D0C]

| [#x0D0E-#x0D10] | [#x0D12-#x0D28] | [#x0D2A-#x0D39]
| [#x0D60-#x0D61] | [#x0E01-#x0E2E] | [#x0E30] | [#x0E32-#x0E33]
| [#x0E40-#x0E45] | [#x0E81-#x0E82] | [#x0E84] | [#x0E87-#x0E88]
| [#x0E8A] | [#x0E8D] | [#x0E94-#x0E97] | [#x0E99-#x0E9F]
| [#x0EA1-#x0EA3] | [#x0EA5] | [#x0EA7] | [#x0EAA-#x0EAB]
| [#x0EAD-#x0EAE] | [#x0EB0] | [#x0EB2-#x0EB3] | [#x0EBD]
| [#x0EC0-#x0EC4] | [#x0F40-#x0F47] | [#x0F49-#x0F69]
| [#x10A0-#x10C5] | [#x10D0-#x10F6] | [#x1100] | [#x1102-#x1103]
| [#x1105-#x1107] | [#x1109] | [#x110B-#x110C] | [#x110E-#x1112]
| [#x113C] | [#x113E] | [#x1140] | [#x114C] | [#x114E] | [#x1150]
| [#x1154-#x1155] | [#x1159] | [#x115F-#x1161] | [#x1163] | [#x1165]
| [#x1167] | [#x1169] | [#x116D-#x116E] | [#x1172-#x1173] | [#x1175]
| [#x119E] | [#x11A8] | [#x11AB] | [#x11AE-#x11AF]
| [#x11B7-#x11B8] | [#x11BA] | [#x11BC-#x11C2] | [#x11EB]
| [#x11F0] | [#x11F9] | [#x1E00-#x1E9B] | [#x1EA0-#x1EF9]
| [#x1F00-#x1F15] | [#x1F18-#x1F1D] | [#x1F20-#x1F45]
| [#x1F48-#x1F4D] | [#x1F50-#x1F57] | [#x1F59] | [#x1F5B]
| [#x1F5D] | [#x1F5F-#x1F7D] | [#x1F80-#x1FB4]
| [#x1FB6-#x1FBC] | [#x1FBE] | [#x1FC2-#x1FC4]
| [#x1FC6-#x1FCC] | [#x1FD0-#x1FD3] | [#x1FD6-#x1FDB]
| [#x1FE0-#x1FEC] | [#x1FF2-#x1FF4] | [#x1FF6-#x1FFC] | [#x2126]
| [#x212A-#x212B] | [#x212E] | [#x2180-#x2182] | [#x3041-#x3094]
| [#x30A1-#x30FA] | [#x3105-#x312C] | [#xAC00-#xD7A3])

[127] Ideographic ::= ([#x4E00-#x9FA5] | [#x3007] | [#x3021-#x3029])

[128] CombiningChar ::= ([#x0300-#x0345] | [#x0360-#x0361] | [#x0483-#x0486]
| [#x0591-#x05A1] | [#x05A3-#x05B9] | [#x05BB-#x05BD]
| [#x05BF] | [#x05C1-#x05C2] | [#x05C4] | [#x064B-#x0652]
| [#x0670] | [#x06D6-#x06DC] | [#x06DD-#x06DF]
| [#x06E0-#x06E4] | [#x06E7-#x06E8] | [#x06EA-#x06ED]
| [#x0901-#x0903] | [#x093C] | [#x093E-#x094C] | [#x094D]
| [#x0951-#x0954] | [#x0962-#x0963] | [#x0981-#x0983] | [#x09BC]
| [#x09BE] | [#x09BF] | [#x09C0-#x09C4] | [#x09C7-#x09C8]
| [#x09CB-#x09CD] | [#x09D7] | [#x09E2-#x09E3] | [#x0A02]
| [#x0A3C] | [#x0A3E] | [#x0A3F] | [#x0A40-#x0A42]
| [#x0A47-#x0A48] | [#x0A4B-#x0A4D] | [#x0A70-#x0A71]
| [#x0A81-#x0A83] | [#x0ABC] | [#x0ABE-#x0AC5]
| [#x0AC7-#x0AC9] | [#x0ACB-#x0ACD] | [#x0B01-#x0B03]
| [#x0B3C] | [#x0B3E-#x0B43] | [#x0B47-#x0B48]
| [#x0B4B-#x0B4D] | [#x0B56-#x0B57] | [#x0B82-#x0B83]
| [#x0BBE-#x0BC2] | [#x0BC6-#x0BC8] | [#x0BCA-#x0BCD]
| [#x0BD7] | [#x0C01-#x0C03] | [#x0C3E-#x0C44] | [#x0C46-#x0C48]
| [#x0C4A-#x0C4D] | [#x0C55-#x0C56] | [#x0C82-#x0C83]
| [#x0CBE-#x0CC4] | [#x0CC6-#x0CC8] | [#x0CCA-#x0CCD]
| [#x0CD5-#x0CD6] | [#x0D02-#x0D03] | [#x0D3E-#x0D43]
| [#x0D46-#x0D48] | [#x0D4A-#x0D4D] | [#x0D57] | [#x0E31]
| [#x0E34-#x0E3A] | [#x0E47-#x0E4E] | [#x0EB1]
| [#x0EB4-#x0EB9] | [#x0EBB-#x0EBC] | [#x0EC8-#x0ECD]
| [#x0F18-#x0F19] | [#x0F35] | [#x0F37] | [#x0F39] | [#x0F3E]

			| [#x0F3F] | [#x0F71-#x0F84] | [#x0F86-#x0F8B] | [#x0F90-#x0F95] | [#x0F97] | [#x0F99-#x0FAD] | [#x0FB1-#x0FB7] | [#x0FB9] | [#x20D0-#x20DC] | [#x20E1] | [#x302A-#x302F] | [#x3099] | [#x309A])
[129]	Digit	::=	([#x0030-#x0039] | [#x0660-#x0669] | [#x06F0-#x06F9] | [#x0966-#x096F] | [#x09E6-#x09EF] | [#x0A66-#x0A6F] | [#x0AE6-#x0AEF] | [#x0B66-#x0B6F] | [#x0BE7-#x0BEF] | [#x0C66-#x0C6F] | [#x0CE6-#x0CEF] | [#x0D66-#x0D6F] | [#x0E50-#x0E59] | [#x0ED0-#x0ED9] | [#x0F20-#x0F29])
[130]	Extender	::=	([#x00B7] | [#x02D0] | [#x02D1] | [#x0387] | [#x0640] | [#x0E46] | [#x0EC6] | [#x3005] | [#x3031-#x3035] | [#x309D-#x309E] | [#x30FC-#x30FE])

B References

B.1 Normative References

XML

World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second edition), October, 2000 See http://www.w3.org/TR/REC-xml.

XML Names

World Wide Web Consortium. Namespaces in XML. W3C Recommendation. See http://www.w3.org/TR/REC-xml-names/ .

XML Schema Part 0

World Wide Web Consortium. XML Schema Part 0 : Primer. W3C Recommendation, May 2001. See http://www.w3.org/TR/xmlschema-0/

XML Schema Part 1

World Wide Web Consortium. XML Schema Part 1 : Structures. W3C Recommendation, May 2001. See http://www.w3.org/TR/xmlschema-1/

XML Schema Part 2

World Wide Web Consortium. XML Schema Part 2 : Datatypes. W3C Recommendation, May 2001. See http://www.w3.org/TR/xmlschema-2/.

XQuery 1.0: A Query Language for XML

World Wide Web Consortium.*XQuery 1.0: A Query Language for XML*. W3C Working Draft, 20 December 2001. See http://www.w3.org/TR/xquery/

XQuery 1.0 and XPath 2.0 Data Model

World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft, 20 December 2001. See http://www.w3.org/TR/query-datamodel/.

XQuery 1.0 and XPath 2.0 Functions and Operators

World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C Working Draft, 20 December 2001. See http://www.w3.org/TR/xquery-operators/

B.2 Non-normative References

XML Path Language (XPath) : Version 1.0

World Wide Web Consortium. XML Path Language (XPath) : Version 1.0. W3C Recommendation, November, 1999. See http://www.w3.org/TR/xpath.html.

XML Path Language (XPath) 2.0

World Wide Web Consortium. XML Path Language (XPath) 2.0. W3C Working Draft, 20 December 2001. See http://www.w3.org/TR/xpath20/.

XML Query 1.0 Requirements

World Wide Web Consortium. *XML Query 1.0 Requirements*. W3C Working Draft, 15 Feb 2001. See http://www.w3.org/TR/xmlquery-req.

XML Query Use Cases

World Wide Web Consortium. *XML Query Use Cases*. W3C Working Draft, 20 Dec 2001. See http://www.w3.org/TR/xmlquery-use-cases.

XML Schema : Formal Description

World Wide Web Consortium. XML Schema: Formal Description. W3C Working Draft, March 2001. See http://www.w3.org/TR/xmlschema-formal/

XSLT 99

World Wide Web Consortium. XSL Transformations (XSLT), Version 1.0. W3C Recommendation, November 1999. See http://www.w3.org/TR/xslt.

B.3 Background References

BFS00

P. Buneman, M. Fernandez, D. Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, April, 2000, Vol 9, Number 1.

BKD90

Francois Bancilhon, Paris Kanellakis, Claude Delobel. *Building an Object-Oriented Database System*. Morgan Kaufmann, 1990.

BNTW95

Peter Buneman, Shamim Naqvi, Val Tannen, Limsoon Wong. Principles of programming with complex object and collection types. *Theoretical Computer Science* 149(1):3--48, 1995.

CM93

S. Cluet and G. Moerkotte. Nested queries in object bases. *Workshop on Database Programming Languages*, pages 226--242, New York, August 1993.

Col90

L. S. Colby. A recursive algebra for nested relations. *Information Systems* 15(5):567-582, 1990.

Graefe93

Goetz Graefe, *Query Evaluation Techniques for Large Databases*. In *ACM Computing Surveys*, 25(2):73--170, 1993.

HP2000

Haruio Hosoya, Benjamin Pierce, XDuce : A Typed XML Processing Language (Preliminary Report)

WebDB Workshop 2000.

Languages

Handbook of Formal Languages. G. Rozenberg and A. Salomaa, editors. *Springer-Verlag*. 1997.

LMW96

Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multi-dimensional arrays: Design, implementation, and optimization techniques. *SIGMOD* 1996.

LW97

Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *Journal of Computer and Systems Sciences*, 55(2):241--272, October 1997.

Milner

R. Milner, M. Tofte, R. Harper, D. MacQueen *The Definition of Standard ML (Revise)*. MIT Press, 1997.

Mitchell

John C. Mitchell *Foundations for Programming Languages*. MIT Press, 1998.

Mog89

E. Moggi, Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science* Asilomar, California, IEEE, June 1989.

Mog91

E. Moggi, Notions of computation and monads. *Information and Computation*, 93(1), 1991.

ODMG

Rick Cattell et al. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann Publishers, San Francisco, 1996.

Quilt

Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. *International Workshop on the Web and Databases (WebDB'2000)*, Dallas, Texas, May 2000.

SQL

International Organization for Standardization (ISO). *Information Technology-Database Language SQL*. Standard No. ISO/IEC 9075:1999. (Available from American National Standards Institute, New York, NY 10036, (212) 642-4900.)

TATA

Tree Automata Techniques and Applications. H. Comon and M. Dauchet and R. Gilleron and F. Jacquemard and D. Lugiez and S. Tison and M. Tommasi. See http://www.grappa.univ-lille3.fr/tata/. 1997.

Wad93

P. Wadler, Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series, Springer Verlag, 1993. Also in J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, Springer Verlag, 1995.

Wad95

P. Wadler, How to declare an imperative. *ACM Computing Surveys*, 29(3):240--263, September 1997.

Won00

Limsoon Wong. An introduction to the Kleisli query system and a commentary on the influence of functional programming on its implementation. *Journal of Functional Programming*, to appear.

XMLQL99

A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. *A query language for XML*. In *International World Wide Web Conference*, 1999.

XQL99

J. Robie, editor. XQL '99 Proposal, 1999. See http://www.ibiblio.org/xql/xql-proposal.html.

YAT99

S. Cluet, and J. Siméon. YATL: A Functional and Declarative Language for XML. See http://db.bell-labs.com/user/simeon/icfp.ps

C Issues

C.1 Introduction

The issues in **[C.2 Issues list]** serve as a design history for this document. The ordering of issues is irrelevant. Each issue has a unique id of the form Issue-<dddd> (where d is a digit). This can be used for referring to the issue by <url-of-this-document>#Issue-<dddd>. Furthermore, each issue has a mnemonic header, a date, an optional description, and an optional resolution. For convenience, resolved issues are displayed in green. Some of the descriptions of the resolved issues are obsolete w.r.t. to the current version of the document.

> **Ed. Note:** Peter (Aug-05-2000): For the sake of archival, there are some duplicate issues raised in multiple instances. Duplicate issues are marked as "resolved" with reference to the representative issue.

C.2 Issues list

Issue-0001: Attributes

Date: Jul-26-2000
Raised by: Algebra Editors

Description: One example of the need for support of **[Issue-0049: Unordered Collections]**, but also: Attributes need to be constrained to contain white space separated lists of simple types only.

Resolution: Attributes are represented by attribute attribute-name { content }.

Issue-0002: Namespaces

Date: Jul-26-2000
Raised by: Algebra Editors

Resolution: Namespaces are represented by {uri-of-namespace}localname.

Issue-0003: Document Order

Date: Jul-26-2000
Raised by: Algebra Editors

Description: The data model and algebra do not define a global order on documents. Querying global order is often required in document-oriented queries.

Resolution: Resolved by adding < operator defined on nodes in same document. See **[Issue-0079: Global order between nodes in different documents]** for order between nodes in different documents.

Issue-0004: References vs containment

Date: Jul-26-2000
Raised by: Algebra Editors

Description: The query-algebra datamodel currently does not explicitly model children-elements by references (other than the XML-Query Datamodel. This facilitates presentation, but may be an oversimplification with regard to **[Issue-0005: Element identity]**.

Resolution: This issue is resolved by subsumption as follows: (1) All child-elements are (implicit) references to nodes. (2) Thus, having resolved **[Issue-0005: Element identity]** this issue is resolved too.

Issue-0005: Element identity

Date: Jul-26-2000
Raised by: Algebra Editors

Description: Do expressions preserve element identity or don't they? And does "=" and distinct use comparison by reference or comparison by value?

Resolution: The first part of the question has been resolved by resolution of **[Issue-0010: Construct values by copy]**. The second part raises a more specific issue **[Issue-0066: Shallow or Deep Equality?]**.

Issue-0006: Source and join syntax instead of "for"

Date: Jul-26-2000
Raised by: Algebra Editors

Description: Another term for "source and join syntax" is "comprehension".

Resolution: This issue is resolved by subsumption under **[Issue-0021: Syntax]**. List comprehension is a syntactic alternative to "for v in e1 do e2", which has been favored by the WG in the resolution of **[Issue-0021: Syntax]**.

Issue-0007: References: IDREFS, Keyrefs, Joins

Date: Jul-26-2000
Raised by: Algebra Editors

Description: Currently, the Algebra does not support reference values, such as IDREF, or Keyref (not to be mixed up with "node-references" - see **[Issue-0005: Element identity]**, which are defined in the XML Query Data Model. The Algebra's type system should be extended to support reference types and the data model operators `ref`, and `deref` should be supported (similar to id() in XPath).

Resolution: Delegated to XPath 2.0. Algebra should adopt solutions (e.g., id()/keyref() functions) provided in XPath 2.0. There may be an interaction between IDREFs and RefNodes, but we're not going to cover that now.

Issue-0008: Fixed point operator or recursive functions

Date: Jul-26-2000
Raised by: Algebra Editors

Description: It may be useful to add a fixed-point operator, which can be used in lieu of recursive functions to compute, for example, the transitive closure of a collection.

Currently, the Algebra does not guarantee termination of recursive expressions. In order to ensure termination, we might require that a recursive function take one argument that is a singleton element, and any recursive invocation should be on a descendant of that element; since any element has a finite number of descendants, this avoids infinite regress. (Ideally, we should have a simple syntactic rule that enforces this restriction, but we have not yet devised such a rule.)

Impacts optimization; hard to do static type inference; current algebra is first-order

Issue-0009: Externally defined functions

Date: Jul-26-2000
Raised by: Algebra Editors

Description: There is no explicit support for externally defined functions.

The set of built-in functions may be extended to support other important operators.

Resolution: Algebra editors endorse a solution that uses XP for specifying signatures of external functions. Algebra will adopt solution provided by XQuery.

Issue-0010: Construct values by copy

Date: Jul-26-2000
Raised by: Algebra Editors

Description: Need to be able to construct new types from bits of old types by reference and by copy. Related to **[Issue-0005: Element identity]**.

Resolution: The WG wishes to support both: construction of values by copy, as well as references to original nodes. This needs some further investigation to sort out all technical difficulties (see **[Issue-0062: Open questions for constructing elements by reference]**) so the change has not yet been reflected in the Algebra document.

Issue-0011: XPath tumbler syntax instead of index?

Date: Jul-26-2000
Raised by: Algebra Editors

Description: XPath provides as a shorthand syntax [integer] to select child-elements by their position on the sibling axes, whereas the xml-query algebra uses a combination of a built-in function index() and iteration.

Addendum by JS (submitted by MF) Dec 19/2000: The typing of index is lossy : it produces a factored type. Jerome suggests the more precise `range` operator:

```
e : q min m  max n   n' - (m'-1) = r  m' >= m   n' <= n
-----------------------------------------------
```

```
            range(e;m';n') : q min r  max r

nth(e;n) == range(e;n;n)
```

The `range` operator takes a repetition of prime types and those values in the range `m'` to `n'`; if the repetition does not include that range, a run-time error is raised. The `range` and `nth` operators could also be defined in terms of `head` and `tail` and polymorphic recursive functions. In the absence of parameteric polymorphism, it is not possible to define `range` and `nth` with precise types.

Here are Peter's rules:

```
e : p min m  max n     n!=*
-------------------------------------------------
range(e;m';n') : p{n'-max(m,m')+1,min(n',n)-m'+1}

For example:
let v1 = a[] min 2  max 4

range(v1;3;3): a[] min 1  max 1
range(v1;1;3): a[] min 2  max 3
range(v1;3;5): a[] min 1  max 2
range(v1;1;5): a[] min 2  max 4

e : p min m  max *
-----------------------------
range(e;m';n') : p min 0  max n'-m'+1

let v2 = a[] min 0  max *

range(v2;1;3): a[] min 0  max 2

this follows the typical semantics for head() and tail():
head(()) = tail(()) = ()
and the semantics behind
range(e;m',n') = tail o ...(m' times)   ... o tail o head,
                 tail o ...(m'+1 times) ... o tail o head,
                 ...
                 tail o ...(n' times)   ... o tail o head
```

I would have no troubles in restricting ourselves to nth() instead of range() in the algebra (range can always be enumerated by nth()). Furthermore, we should consider whether m',n' can be computed numbers.

Issue-0012: GroupBy - needs second order functions?

Date: Jul-26-2000
Raised by: Algebra Editors

Description: The type system is currently first order: it does not support function types nor higher-order functions. Higher-order functions are useful for specifying, for example, sorting and grouping operators, which take other functions as arguments.

Resolution: The WG has decided to express groupBy by a combination of `for` and `distinct`. Thus w.r.t. to GroupBy this Issue is resolved. Because GroupBy is not the only use case for higher order functions, a new issue **[Issue-0063: Do we need (user defined) higher order functions?]** is raised.

Issue-0013: Collations

Date: Jul-26-2000
Raised by: Algebra Editors

Description: Collations identify the ordering to be applied for sorting strings. Currently, it is considered to have an (optional parameter) collation "name" as follows: "SORT variable IN exp BY +(expression {ASCENDING|DESCENDING} {COLLATION name}). An alternative would be to model a collation as a simple type derived from string, and use type-level casting, i.e. expression :collationtype (which is already supported in the XML Query Algebra), for specifying the collation. That would make: "SORT variable IN exp BY +(expression:collationname {ASCENDING|DESCENDING}). But that requires some support from XML-Schema.

More generally, collations are important for any operator in the Algebra that involves string comparison, among them: sort, distinct, "=" and "<".

Resolution: Formal semantics will adopt solution provided by Operators.

Issue-0014: Polymorphic types

Date: Jul-26-2000
Raised by: Algebra Editors

Description: The type system is currently monomorphic: it does not permit the definition of a function over generalized types. Polymorphic functions are useful for factoring equivalent functions, each of which operate on a fixed type.

The current type system has already a built-in polymorphic type (lists) and is likely to have more (unordered collections). The question is, whether to allow for user-defined polymorphic types and user defined polymorphic functions.

Issue-0015: 3-valued logic to support NULLs

Date: Jul-26-2000
Raised by: Algebra Editors

Resolution: The Formal Semantics supports the current semantics of NULL values, as described in the XQuery December working draft. The Formal Semantics will reflect further resolution of open issues on NULLs and 3 valued logic as decided by XQuery.

Issue-0016: Mixed content

Date: Jul-26-2000
Raised by: Algebra Editors

Description: The XML-Query Algebra allows to generate elements with an arbitrary mixture of data (of simple type) and elements. XML-Schema only allows for a combination of strings interspersed with elements (aka mixed content). We need to figure out whether and how to constrain the XML-Query Algebra accordingly (e.g. by typing rules?)

Resolution: The type system has been extended to support the interleaving operator & - see **[3 The XQuery Type System]**. Mixed content is defined in terms of &.

Issue-0017: Unordered content

Date: Jul-26-2000
Raised by: Algebra Editors

Description: All-groups in XML-Schema, not to be mixed up with **[Issue-0049: Unordered Collections]**

Resolution: The type system has been extended with the support of all-groups - see **[3 The XQuery Type System]**.

Issue-0018: Align algebra types with schema

Date: Jul-26-2000
Raised by: Algebra Editors

Description: The Algebra's internal type system is the type system of XDuce. A potentially significant problem is that the Algebra's types may lose information when converted into XML Schema types, for example, when a result is serialized into an XML document and XML Schema.

James Clark points out : "The definition of AnyComplexType doesn't match the concrete syntax for types since it applies unbounded repetition to AnyTree and one alternative for AnyTree is AnyAttribute." This is another example of an alignment issue.

This issue comprises also issues **[Issue-0016: Mixed content]**, **[Issue-0017: Unordered content]**, **[Issue-0053: Global vs. local elements]**, **[Issue-0054: Global vs. local complex types]**, **[Issue-0019: Support derived types]**, substitution groups.

Issue-0019: Support derived types

Date: Jul-26-2000
Raised by: Algebra Editors

Description: The current type system does not support user defined type hierarchies (by extension or by restriction).

Issue-0020: Structural vs. name equivalence

Date: Jul-26-2000
Raised by: Algebra Editors

Description: The subtyping rules in **[3.3 Subtyping]** only define structural subtyping. We need to extend this with support for subtyping via user defined type hierarchies - this is related to **[Issue-0019: Support derived types]**.

Issue-0021: Syntax

Date: Jul-26-2000
Raised by: Algebra Editors

Description: (e.g. for.<-.in vs for.in.do)

Resolution: The WG has voted for several syntax changes , "for v in e do e", "let v = e do", "sort v in e by e ...", "distinct", "match case v:t e ... else e".

Issue-0022: Indentation, Whitespaces

Date: Jul-26-2000
Raised by: Algebra Editors

Description: Is indentation significant?

Resolution: The WG has consensus that indentation is not significant , i.e., all documents are white space normalized.

Issue-0023: Catch exceptions and process in algebra?

Date: Jul-26-2000
Raised by: Algebra Editors

Description: Does the Algebra give explicit support for catching exceptions and processing them?

Resolution: Subsumed by new issue **[Issue-0064: Error code handling in Query Algebra]**.

Issue-0024: Value for empty sequences

Date: Jul-26-2000
Raised by: Algebra Editors

Description: What does "value" do with empty sequences?

Resolution: The definition of value(e) has changed to:

```
value(e) = typeswitch children(e)
                    case v: AnyScalar do v
                    else()
```

Furthermore, the typing rules for "for v in e1 do e2" have been changed such that the variable v is typed-checked seperately for each unit-type occuring in expression e1.

Consequently the following example would be typed as follows:

```
query for b in b0/book do
      value(b/year): xs:integer min 0 max *
```

rather than leading to an error.

Issue-0025: Treatment of empty results at type level

Date: Jul-26-2000
Raised by: Algebra Editors

Description: This is related to **[Issue-0024: Value for empty sequences]**.

Resolution: Resolved by resolution of **[Issue-0025: Treatment of empty results at type level]**.

Issue-0026: Project - one tag only

Date: Jul-26-2000

Raised by: Algebra Editors

Description: Project is only parameterized by one tag. How can we translate a0/(b | c)?

Resolution: With the new syntax (and type system) a0/(b | c) can be translated to "for v in a0 do typeswitch case v1:b[AnyType] do v1 case v2:c[AnyType] do c else ()".

Issue-0027: Case syntax

Date: Jul-26-2000
Raised by: Quilt Comments et al.

Description: N-ary case can be realized by nested binary cases.

Resolution: New (n-ary) case syntax is introduced.

Issue-0028: Fusion

Date: Jul-26-2000
Raised by: Michael Rys

Description: Does the Algebra support fusion as introduced by query languages such as LOREL? This is related to **[Issue-0005: Element identity]**, because fusion only makes sense with support of element identity.

Resolution: Fusion is equivalent to 'natural full-outer join'. XQuery can reraise issue if desired. If added, the Algebra editors should review any solution w.r.t typing.

Issue-0029: Views

Date: Jul-26-2000
Raised by: Michael Rys

Description: One of the problems in views: Can we undeclare/hide things in environment? For example, if we support element-identity, can we explicitly discard a parent, and/or children from an element in the result-set? Related to **[Issue-0005: Element identity]**.

Resolution: XQuery can reraise issue if desired. If added, the Algebra editors should review any solution w.r.t typing.

Issue-0030: Automatic type coercion

Date: Jul-26-2000
Raised by: Dana Florescu

Description: What do we do if a value does not have a type or a different type from what is required?

Suggested Resolution: We believe that the XML Query Language should specify default type coercions for mixed mode arithmetic should be performed according to a fixed precedence hierarchy of types, specifically integer to fixed decimal, fixed decimal to float, float to double. This policy has the advantage of simplicity, tradition, and static type inference. Programmers could explicitly specify alternative type coercions when desirable.

Resolution: Delegation to XPath 2.0, XQuery, and/or Operators.

Issue-0031: Recursive functions

Date: Jul-26-2000
Raised by: Dana Florescu

Resolution: subsumed by [**Issue-0008: Fixed point operator or recursive functions**]

Issue-0032: Full regular path expressions

Date: Jul-26-2000
Raised by: Dana Florescu

Description: Full regular path expressions allow to constrain recursive navigation along paths by means of regular expressions, e.g. a/b*/c denotes all paths starting with an a, proceeding with arbitrarily many b's and ending in a c. Currently the XML-Query Algebra can express this by means of (structurally) recursive functions. An alternative may be the introduction of a fixpoint operator [**Issue-0008: Fixed point operator or recursive functions**].

Resolution: XPath 2.0 can raise issue if desired. The Algebra editors should review any solution w.r.t typing.

Issue-0033: Metadata Queries

Date: Jul-26-2000
Raised by: Dana Florescu

Description: Metadata queries are queries that require runtime access to type information.

Issue-0034: Fusion

Date: Jul-26-2000
Raised by: Dana Florescu

Resolution: Identical with [**Issue-0028: Fusion**]

Issue-0035: Exception handling

Date: Jul-26-2000
Raised by: Dana Florescu

Resolution: Subsumed by [**Issue-0023: Catch exceptions and process in algebra?**] and [**Issue-0064: Error code handling in Query Algebra**].

Issue-0036: Global-order based operators

Date: Jul-26-2000
Raised by: Dana Florescu

Resolution: Subsumed by [**Issue-0003: Document Order**]

Issue-0037: Copy vs identity semantics

Date: Jul-26-2000
Raised by: Dana Florescu

Resolution: subsumed by [**Issue-0005: Element identity**]

Issue-0038: Copy by reachability

Date: Jul-26-2000
Raised by: Dana Florescu

Description: Is it possible to copy children as well as IDREFs, Links, etc.? Related to **[Issue-0005: Element identity]** and **[Issue-0008: Fixed point operator or recursive functions]**

Resolution: Resolved by addition of "deep" copy operator in [XQuery 1.0 and XPath 2.0 Data Model].

Issue-0039: Dereferencing semantics

Date: Jul-26-2000
Raised by: Dana Florescu

Resolution: Subsumed by **[Issue-0005: Element identity]**

Issue-0040: Case Syntax

Date: Aug-01-2000
Raised by: Quilt

Description: We suggest that the syntax for "case" be made more regular. At present, it takes only two branches, the first labelled with a tag-name and the second labelled with a variable. A more traditional syntax for "case" would have multiple branches and label them in a uniform way. If the algebra is intended only for semantic specification, "case" may not even be necessary.

Resolution: subsumed by **[Issue-0027: Case syntax]**

Issue-0041: Sorting

Date: Aug-01-2000
Raised by: Quilt

Description: We are not happy about the three-step sorting process in the Algebra. We would prefer a one-step sorting operator such as the one illustrated below, which handles multiple sort keys and mixed sorting directions: SORT emp <- employees BY emp/deptno ASCENDING emp/salary DESCENDING

Resolution: The WG has decided to go for the above syntax, with an (optional) indication of COLLATION.

Issue-0042: GroupBy

Date: Aug-01-2000
Raised by: Quilt

Description: We do not think the algebra needs an explicit grouping operator. Quilt and other high-level languages perform grouping by nested iteration. The algebra can do the same.

related to **[Issue-0012: GroupBy - needs second order functions?]**

Resolution: The WG has decided to skip groupBy for the time being.

Issue-0043: Recursive Descent for XPath

Date: Aug-01-2000
Raised by: Quilt

Description: The very important XPath operator "//" is supported in the Algebra only by writing a recursive

function. This is adequate for a semantic specification, but if the Algebra is intended as an optimizable target language it will need better support for "//" (possibly in the form of a fix-point operator.)

Resolution: Resolved by subsumption under **[Issue-0043: Recursive Descent for XPath]**

Issue-0044: Keys and IDREF

Date: Aug-01-2000
Raised by: Quilt

Description: We think the algebra needs some facility for dereferencing keys and IDREFs (exploiting information in the schema.)

Resolution: Subsumed by **[Issue-0007: References: IDREFS, Keyrefs, Joins]**

Issue-0045: Global Order

Date: Aug-01-2000
Raised by: Quilt

Description: We are concerned about absence of support for operators based on global document ordering such as BEFORE and AFTER.

Resolution: Subsumed by **[Issue-0003: Document Order]**

Issue-0046: FOR Syntax

Date: Aug-01-2000
Raised by: Quilt

Description: We agree with comments made in the face-to-face meeting about the aesthetics of the Algebra's syntax for iteration. For example, the following syntax is relatively easy to understand: FOR x IN some_expr EVAL f(x) whereas we find the current algebra equivalent to be confusing and misleading: FOR x <- some_expr IN f(x) This syntax appears to assign the result of some_expr to variable x, and uses the word IN in a non-intuitive way.

Resolution: Subsumed by **[Issue-0021: Syntax]**

Issue-0047: Attributes

Date: Aug-01-2000
Raised by: Quilt

Description: See **[Issue-0001: Attributes]**.

Resolution: Subsumed by **[Issue-0001: Attributes]**

Issue-0048: Explicit Type Declarations

Date: Jul-27-2000
Raised by: Group 1 at F2F, Redmond

Description: Type Declaration for the results of a query: The issue is whether to auto construct the result type from a query or to pre-declare the type of the result from a query and check for correct type on the return value. Suggestion: Support for pre-declared result data type and as well as to coerce the output to a new type is

desirable. Runtime or compile time type checking is to be resolved? Once you attach a name to a type, it is preserved during the query processing.

Resolution: W.r.t. compile time type casts this is already possible with e:t. For run-time casts an issue has been raised in **[Issue-0062: Open questions for constructing elements by reference]**.

Issue-0049: Unordered Collections

Date: Jul-27-2000
Raised by: Algebra Editors, Group 1, F2F, Redmond

Description: Currently, all sequences in the data model are ordered. It may be useful to have unordered forests. The `distinct-node` function, for example, produces an inherently unordered forest. Unordered forests can benefit from many optimizations for the relational algebra, such as commutable joins.

Handling of collection of attributes is easy but the collection of elements is complex due to complex type support for the elements. It makes sense to allow casting from unordered to ordered collection and vice versa. It is not clear whether the new ordered or unordered collection is a new type or not. It affects function resolution, optimization.

Our request to Schema to represent insignificance of ordering at schema level has not been fulfilled. Thus we need to be aware that this information may get lost, when mapping to schema.

Resolution: Unordered collections are described by {t} see **[3 The XQuery Type System]**, some operators (sort, distinct-node, for, and sequence) are overloaded, and some operators (difference, intersection) are added). A new issue **[Issue-0076: Unordered types]** is raised.

Issue-0050: Recursive Descent for XPath

Date: Jul-27-2000
Raised by: Group 1, F2F, Redmond

Description: Suggestion: The group likes to add a support for fixed-point operator in the query language that will allow us to express the semantics of the // operator in an xpath expression. A path expression of the form a//b may be represented by a fixed-point operator fp(a, "/.")/b.

Resolution: Subsumed by **[Issue-0043: Recursive Descent for XPath]**

Issue-0051: Project redundant?

Date: Aug-05-2000
Raised by: Peter Fankhauser

Description: It appears that project a e could be reduced to sth. like

```
for v <- e in case v of a[v1] =>
        a[v1] | v2 => ()
```

... or would that generate a less precise type?

Resolution: With the new type system and handling of the for operator, project is indeed redundant.

Issue-0052: Axes of XPath

Date: Aug-05-2000

Raised by: Peter Fankhauser

Description: The current algebra makes navigation to parents difficult to impossible. With support of Element Identity **[Issue-0005: Element identity]** and recursive functions **[Issue-0008: Fixed point operator or recursive functions]** one can express parent() by a recursive function via the document root. More direct support needs to be investigated w.r.t its effect on the type system.

The WG wishes to support a built-in operator parent().

Resolution: XPath 2.0 and XQuery can reraise issue if desired. Algebra should review any solution w.r.t typing. Question: whether namespace axis (i.e., access namespace nodes) will be included in XQuery. Algebra currently has issues related to typing of parent() and descendant(). If sibling axes are included in XQuery, then Algebra should review w.r.t. typing.

Issue-0053: Global vs. local elements

Date: Aug-05-2000
Raised by: Peter Fankhauser

Description: The current type system cannot represent global element-declarations of XML-Schema. All element declarations are local.

Resolution: The type system now supports both local and global elements and attributes.

Issue-0054: Global vs. local complex types

Date: Aug-05-2000
Raised by: Peter Fankhauser

Description: The current type system does not distinguish between global and local types as XML-Schema does. All types appear to be fully nested (i.e. local types)

Resolution: The type system now supports both local and global types.

Issue-0055: Types with non-wellformed instances

Date: Aug-05-2000
Raised by: Peter Fankhauser

Description: The type system and algebra allows for sequences of simple types, which can usually be not represented as a well-formed document. How shall we constrain this? Related to **[Issue-0016: Mixed content]**.

Issue-0056: Operators on Simple Types

Date: Jul-15-2000
Raised by: Fernandez et al.

Description: We intentionally did not define equality or relational operators on element and simple type. These operators should be defined by consensus.

> **Ed. Note:** MF, 15-Jan-2001 A joint task force on operators with members from the XSLT, XML Schema, and XML Query working groups is chartered to define arithmetic operators.

Resolution: XQuery formal semantics adopts solution provided by Operators task force.

Issue-0057: More precise type system; choice in path

Date: Aug-07-2000
Raised by: LA-Team

Description: (This subsumes **[Issue-0051: Project redundant?]**). If the type system were more precise, then (project a e) could be replaced by:

```
for v &lt;- e in
    case v of
      a[v1] =&gt; a[v1]
    | v2 =&gt; ()
```

One could also represent (e/(a|b)) directly in a similar style.

```
for v &lt;- e in
    case v of
      a[v1] =&gt; a[v1]
    | v2 =&gt; case v2 of
      b[v3] =&gt; b[v3]
            | v4 =&gt; ()
```

Currently, there is no way to represent (e/(a|b)) without loss of precision, so if we do not change the type system, we may need to have some way to represent (e/(a|b)) and similar terms without losing precision. (The LA team has a design for this more precise type system, but it is too large to fit in the margin of this web page!)

Resolution: See resolution of **[Issue-0051: Project redundant?]**

Issue-0058: Downward Navigation only?

Date: Aug-07-2000
Raised by: LA-Team

Description: Related to **[Issue-0052: Axes of XPath]**. The current type system (and the more precise system alluded to in **[Issue-0057: More precise type system; choice in path]**) seems well suited for handling XPath children and descendant axes, but not parent, ancestor, sibling, preceding, or following axes. Is this limitation one we can live with?

Resolution: Subsumed by **[Issue-0052: Axes of XPath]**

Issue-0059: Testing Subtyping

Date: Aug-07-2000
Raised by: LA-Team

Description: One operation required in the Algebra is to test whether XML type t1 is a subtype of XML type t2, indicated by writing t1 <: t2. There is a well-known algorithm for this, based on tree automata, which is a straightforward variant of the well-known algorithm for testing whether the language generated by one regular-expression is a subset of the language generated by another. (The algorithm involves generating deterministic automata for both regular expressions or types.)

However, the naive implementation of the algorithm for comparing XML types can be slow in practice, whereas the naive algorithm for regular expressions is tolerably fast. The only acceptably fast implementation

of a comparison for XML types that the LA team knows of has been implemented by Haruo Hasoya, Jerome Voullion, and Benjamin Pierce at the University of Pennsylvania, for their implementation of Xduce. (Our implementation of the Algebra re-uses their code, with permission.)

So, should we adopt a simpler definition of subtyping which is easier to test? One possibility is to adopt the sibling restriction from Schema, which requires that any two elements which appear a siblings in the same content model must themselves have contents of the same type. Jerome Simeon and Philip Wadler discovered that adopting the sibling restriction reduces the problem of checking subtyping of XML types to that of checking regular languages for inclusion, so it may be worth adopting the restriction for that reason.

Issue-0060: Internationalization aspects for strings

Date: Jun-26-2000
Raised by: I18N

Description: These issues are taken from the comments on the Requirements Document by I18N

Further information can be found at http://www.w3.org/TR/WD-charreq.

It is a goal of i18n that queries involving string matching ("select x where x='some_constant'") treat canonically equivalent strings (in the Unicode sense) as matching. If the query and the target are both XML, early normalization (as per the Character Model) is assumed and binary comparison ensures that the equivalence requirement is satisfied. However, if the target is originally a legacy database which logically has a layer that exports the data as XML, that XML must be exported in normalized form. The XML Query spec must impose the normalization requirement upon such layers.

Similarly, the query may come from a user-interface layer that creates the XML query. The XML Query spec must impose the normalization requirement upon such layers.

Provided that the query and the target are in normalized form C, the output of the query must itself be in normalized form C.

Queries involving string matching should support various kinds of loose matching (such as case-insensitivity, katakana-hiragana equivalence, accent-accentless equivalence, etc.)

If such features as case-insensitivity are present in queries involving string matching, these features must be properly internationalized (e.g. case folding works for accented letters) and language-dependence must be taken into account (e.g. Turkish dotless-i).

Queries involving character counting and indexing must take into account the Character Model. Specifically, they should follow Layer 3 (locale-independent graphemes). Additional details can be found in The Unicode Standard 3.0 and UTR#18. Queries involving word counting and indexing should similarly follow the recommendations in these references.

Resolution: XQuery formal semantics adopts solution provided by Operators task force.

Issue-0061: Model for References

Date: Aug-16-2000
Raised by: Group 3, F2F, Redmond

Description: Related to a number of issues around **[Issue-0005: Element identity]**.

- Use Cases

Table of Contents

REF *could* do this well if it were restructured - it does not maintain unforeseen relationships or use them...

Bibliographies

Recursive parts

RDF assertions

Inversion of simple parent/child references (related to **[Issue-0058: Downward Navigation only?]**).

- What can we leave out?

  can we leave out transitive closure?

  can we limit recursion?

  can we leave out fixed point recursion?

  related to **[Issue-0008: Fixed point operator or recursive functions]**

- Do we need to be able to...

  a. Find the person with the maximum number of descendants?

  b. Airplane routes: how can I get from RDU to Raleigh? (fixed point: guaranteeing termination in reasonable time...)

  c. Given children and their mothers, can I get mothers and their children? (without respect to the form of the original reference...)

  related to **[Issue-0008: Fixed point operator or recursive functions]**.

- Should we abstract out the difference between different kinds of references? If so, should we be able to cast to a particular kind of reference in the output?

  a. abstracting out the differences is cheaper, which is kewl...

  b. the kind of reference gives me useful information about: locality (same document, same repository, big bad internet...) static vs. dynamic (xpointer *may* be resolved dynamically, or *may* be resolved at run time, ID/IDREF is static).

  related to **[Issue-0007: References: IDREFS, Keyrefs, Joins]**.

- do we need to be able to generate ids, e.g. using skolem functions?

  for a document in RAM, or in a persistent tree, identity may be present, implicit, system dependent, and cheap - it's nice to have an abstraction that requires no more than the implicit identity

  persistable ID is more expensive, may want to be able to serialize with ID/IDREF to instantiate references in the data model

  can use XPath instead of generating ID/IDREF, but these references are fragile, and one reason for queries is to create data that may be processed further

persistable ID unique within a repository context

persistable ID that is globally unique

related to [**Issue-0005: Element identity**].

- copy vs. reference semantics

  "MUST not preclude updates..."

  in a pure query environment, sans update, we do not need to distinguish these

  if we have update, we may need to distinguish, perhaps in a manner similar to "updatable cursors" in SQL

  programs may do queries to get DOM nodes that can that be modified. It is essential to be able to distinguish copies of nodes from the nodes themselves.

  copy semantics - what does it mean?

  copy the descendant hierarchy?

  copy the reachability tree? (to avoid dangling references)

  related to [**Issue-0038: Copy by reachability**].

Resolution: Handled in current data model and algebra.

The following issues have been raised since Sep-25-2000.

Issue-0062: Open questions for constructing elements by reference

Date: Sep-25-2000
Raised by: Mary Fernandez et al.

Description: (1) What is the value of parent() when constructing new elements with children refering to original nodes?

(2) Is an approach to either make copies for all children or provide references to all children, or should we allow for a more flexible combination of copies and references?

Resolution: Operational semantics specifies that element node constructor creates copies of all its children. Addition of RefNode in [XQuery 1.0 and XPath 2.0 Data Model] supports explicit reference value.

Issue-0063: Do we need (user defined) higher order functions?

Date: Oct-16-2000
Raised by: Peter Fankhauser

Description: The current XML-Query-Algebra does not allow functions to be parameters of another function - so called higher order functions. However, most of the Algebra operators are (built-in) higher functions, taking expressions as an argument ("sort", "for", "case" to name a few). Even a fixpoint operator, "fun f(x)=e, fix f(x) in e" (see also [**Issue-0008: Fixed point operator or recursive functions**]), would be a built-in higher order function.

Resolution: The XML Query Algebra will not support user defined higher order functions. It does support a

number of built-in higher order functions.

Issue-0064: Error code handling in Query Algebra

Date: Oct-04-2000
Raised by: Rezaur Rahman

Description: How do we return an error code from a function defined in current Query algebra. Do we need to create an array (or a structure) to merge the return value and error code to do this. If that is true, it may be inefficient to implement. In order for cleaner and efficient implementation, it may be necessary to allow a function declaration to take a parameter of type "output" and allow it to return an error code as part of the function definition.

Resolution: One does not need to create a structure to combine return values with error codes, provided each operator or function /either/ returns a value /or/ raises an error. The XML-Query Algebra supports means to raise errors, but does not define standard means to catch errors. Raising errors is accomplished by the expression "error" of type Ø (empty choice). Because Ø | t = t, such runtype errors do not influence static typing. The surface syntax and/or detailed specification of operators on simple types (see **[Issue-0056: Operators on Simple Types]**) may choose to differentiate errors into several error-codes.

Issue-0065: Built-In GroupBy?

Date: Oct-16-2000
Raised by: Peter Fankhauser

Description: We may revisit the resolution of **[Issue-0042: GroupBy]** and reintroduce GroupBy along the lines of sort: "group v in e1 by [e2 {collation}]". One reason for this may be that this allows to use collation for deciding about the equality of strings.

Resolution: The WG has decided to close this issue, and for the time being not consider GroupBy as a built-in operator. Furthermore, **[Issue-0013: Collations]** is ammended to deal with collations for all operators involving a comparison of strings.

Issue-0066: Shallow or Deep Equality?

Date: Oct-16-2000
Raised by: Peter Fankhauser

Description: What is the meaning of "=" and "distinct"? Equality of references to nodes or deep equality of data?

Resolution: [XQuery 1.0 and XPath 2.0 Data Model] defines "=" (value equality) and "==" (identity equality) operators. Description of distinct states that it uses "==".

Issue-0067: Runtime Casts

Date: Sep-21-2000
Raised by: ???

Description: In some contexts it may be desirable to cast values at runtime. Such runtime casts lead to an error if a value cannot be cast to a given type.

Resolution: `cast` *e* `:` *t* has been introduced as a reducible operator expressed in terms of `typeswitch`.

Issue-0068: Document Collections

Date: Oct-16-2000
Raised by: Peter Fankhauser

Description: Per our requirements document we are chartered to support document collections. The current XML-Query Algebra deals with single documents only. There are a number of subissues:

(a) Do we need a more elaborate notion of node-references? E.g. pair of (URI of root-node, local node-ref)

(b) Does the namespace mechanism suffice to type collections of nodes from different documents? Probably yes.

(c) Provided (a) and (b) can be settled, will the approach taken for **[Issue-0049: Unordered Collections]** do the rest?

Issue-0069: Organization of Document

Date: Oct-16-2000
Raised by: Peter Fankhauser

Description: The current document belongs more to the genre (scientific) paper than to the genre specification. One may consider the following modifications: (a) reorganize intro to give a short overview and then state the purpose (strongly typed, neutral syntax with formal semantics as a basis for possibly multiple syntaxes, etc.) (compared to version Aug-23, this version has already gone a good deal in this direction). (b) Equip various definitions and type rules with id's. (c) Elaborate appendices on mapping XML-Query-Algebra Model vs. XML-Query-Datamodel, XML-Query-Type System vs. XML-Schema-Type System. (d) Maybe add an appendix on use-case-solutions. The problem is of course: Part of this is a lot of work, and we may not achieve all for the first release.

Resolution: The WG decided to dispose of this issue. The current overall organization of the document is quite adequate, but of course editorial decisions will have to made all the time.

Issue-0070: Stable vs. Unstable Sort/Distinct

Date: Oct-02-2000
Raised by: Steve Tolkin

Description: Should sort (and distinct) be stable on ordered collections, i.e. lists, and unstable on unordered collections (see **[Issue-0049: Unordered Collections]**)?

Resolution: sort and distinct are stable on ordered collections, and unstable on unordered collections.

Issue-0071: Alignment with the XML Query Datamodel

Date: Sep-26-2000
Raised by: Mary Fernandez

Description: Currently, the XML Query Algebra Datamodel does not model PI's and comments.

Resolution: Addition of operational semantics defines relationship of Algebra to Data Model.

Issue-0072: Facet value access in Query Algebra

Date: Oct-04-2000

Raised by: Rezaur Rahman

Description: Each of the date-time data types have facet values as defined by the schema data types draft spec. This problem is general enough to be applied to other simple data types.

The question is : Should we provide access to these facet values on an instance of a particular data types? If so, what type of access? My take is the facets are to be treated like read-only attributes of a data instance and one should have a read access to them.

Issue-0073: Facets for simple types and their role for typechecking

Date: Oct-16-2000
Raised by: Peter Fankhauser

Description: XML-Schema introduces a number of constraining facets http://www.w3.org/TR/xmlschema-2/ for simple types (among them: length, pattern, enumeration, ...). We need to figure out whether and how to use these constraining facets for type-checking.

Issue-0074: Operational semantics for expressions

Date: Nov-16-2000
Raised by: Mary Fernandez

Description: It is necessary to add an operational semantics that formally defines each operator in the Algebra.

Resolution: The new document contains a full specification of the dynamic semantics.

Issue-0075: Overloading user defined functions

Date: Nov-17-2000
Raised by: Don Chamberlain

Description: User defined functions can not be overloaded in the XML Query Algebra, i.e., a function is exclusively identified by its name, and not by its signature. Should this restriction be relaxed and if so - to which extent?

Resolution: No overloading in Query 1.0

Issue-0076: Unordered types

Date: Dec-11-2000
Raised by: Phil Wadler

Description: Currently unorderedness is represented at type level by {t}, and some (built-in) operators are overloaded such they have different semantics (and potentially different return type) depending on their input type. An alternative is to not represent unorderedness at type level, but rather support unordered for, unordered (unstable) sort, unordered (unstable) distinct.

Resolution: Removed unordered types from type system. Added support for unordered operator.

Issue-0077: Interleaved repetition and closure

Date: Dec-12-2000
Raised by: Peter Fankhauser

Description: Regular Languages are closed w.r.t. to the interleaved product. However, they are not closed w.r.t. to interleaved repetition, which can (e.g) generate the 1 degree Dyck language D[1] = () | a D[1] b | D[1] D[1] = (a,b)^{0,*}, and more generally, any language that coordinates cardinalities of individual members from an alphabeth: E.g. (a ^ b)^ min 0 max * = all strings with equally many a's and b's. These are beyond regular languages. Should we thus try to do without interleaved repetition?

Resolution: if we use interleaved repetition (which we will because it is in MSL), they will be restricted to prime types.

Issue-0078: Generation of ambiguous types

Date: Dec-12-2000
Raised by: Jerome Simeon

Description: Unambiguous content-models in XML 1.0 and XML Schema are not closed w.r.t. union. It appears that the XML Query-Algebra can generate result types which can not be transformed to an unambiguous content-model.

Issue-0079: Global order between nodes in different documents

Date: Dec-16-2000
Raised by: Algebra Editors

Description: The global order operator < is defined on nodes in the same document, but not between nodes in different documents.

Resolution: Resolution follows from the XQuery Data Model. Order between documents is implementation defined but stable.

Issue-0080: Typing of parent

Date: Dec-16-2000
Raised by: Algebra Editors

Description: Currently, the `parent` operator yields an imprecise type : `AnyElement min 0 max 1`. It might be possible to type `parent` more precisely, for example, by using the normalized names in MSL, which encode containment of types.

Issue-0081: Lexical representation of Schema simple types

Date: Jan-17-2001
Raised by: Algebra Editors

Description: Schema simple types must be defined for the Algebra and XQuery.

Resolution: Algebra will adopt lexical reps supported by XQuery.

Issue-0082: Type and expression operator precedence

Date: Jan-17-2001
Raised by: Algebra Editors

Description: The precedence of the type expressions is not defined.

Issue-0083: Expressive power and complexity of typeswitch expression

Date: Jan-17-2001
Raised by: Algebra Editors, Michael Brundage

Description: When processing an XML document without schema information, i.e., the type of the document is AnyComplexType, then match expressions may be very expensive to evaluate:

```
typeswitch x
case t1 : AnyTree do 1
case t2 : AnyTree min 0 max 2 do 2
case t3 : *[*[*[*[* ... [AnyAttribute] ]]]] do 3
else ERROR
```

typeswitch itself is not the issue. The real problem is having very liberal type patterns. We could restrict the kinds of type patterns that we permit.

Resolution: Typeswitch types are now restricted to datatypes. Named typing will further help in reducing the complexity by allowing type annotation contained in the data model as a means for optimization.

Issue-0084: Execution model

Date: Jan-17-2001
Raised by: Algebra Editors

Description: Need prose describing execution model scenarios : interpretor vs. compile/runtime vs. translation into another query language. Explain relationship between static and dynamic semantics.

Resolution: Section **[2.1 Processing model]** defines a procesing model which serves as a framework for the Formal Semantics specification.

Issue-0085: Semantics of Wildcard type

Date: Jan-17-2001
Raised by: Algebra Editors, Michael Brundage

Description: Cite: wildcard types cannot be implemented. If x!y means any name in x except names in y, what does x!y!z mean? In general, how do ! and | operate (precedence, associativity)? Parentheses are required to force the desired grouping of these two operators. Also, what does x!* mean? (There's an infinite family of such examples.)

Resolution: The XQuery type systyem now uses only simple wildcard names based on XPath's NameTest production.

Issue-0086: Syntactic rules

Date: Jan-17-2001
Raised by: Algebra Editors

Description: Need rules for specifying syntactic correctness of query: symbol spaces; variable def'ns precede uses; list of keywords, etc.

Resolution: Syntactic rules should be dealt with in XQuery document

Issue-0087: More examples of Joins

Date: Jan-17-2001
Raised by: Algebra Editors, Michael Brundage

Description: Cite: no join operator; wants example of many-to-many joins, inner join, left and full outer joins.

Resolution: The XQuery document gives a number of such examples.

Issue-0088: Align types with XML Schema : Formal Description.

Date: 02-Apr-2001
Raised by: Mary Fernandez

Description: Sources of misalignment: XQuery types include comment and processing instruction; [XML Schema : Formal Description] does not. XQuery uses () for empty sequence; MSL uses the epsilon character. XQuery permits the names of attribute and element components to be wildcard expressions. MSL only permits literal names for attributes and elements, but permits stand-alone wildcard expressions. XQuery types call '&' interleaved repetition, but MSL says it means 'all g1 and g2 in either order'. Does MSL mean interleaved repetition?

Issue-0089: Syntax for types in XQuery

Date: 30-Apr-2001
Raised by: Mary Fernandez

Description: Formalism document gives a particular syntax for type expressions that is not supported in the XQuery surface syntax.

Issue-0090: Static type-assertion expression

Date: 30-Apr-2001
Raised by: Mary Fernandez

Description: Formalism document uses a static type-assertion expression that is not supported in the XQuery surface syntax.

Resolution: Static type assertion is supported in XQuery with the new "assert as" expression.

Issue-0091: Attribute expression

Date: 30-Apr-2001
Raised by: Mary Fernandez

Description: XQuery formal semantics has stand-alone attribute constructor/expression `ATTRIBUTE QName (Exp)` that is not supported in XQuery surface syntax.

Resolution: Stand-alone attribute construction is supported in XQuery with the new syntax for element and attribute constructors.

Issue-0092: Error expression

Date: 11-May-2001
Raised by: Jerome Simeon

Description: XQuery formal semantics has an error expression `Error` that is not supported in XQuery

surface syntax.

Resolution: Errors are raised by a function "dm:erorr()" instead of a separate expression.

Issue-0093: Representation of Text Nodes in type system

Date: 11-May-2001
Raised by: Mary Fernandez

Description: The data model distinguished between text nodes and strings, which are simple-typed values. Text nodes have identity, parents, and siblings. Strings do not. Text nodes are accessed by the children() accessor; strings and other simple-typed values are accessed by the typed-value() accessor. The distinction between text nodes and simple-typed values should exist in type system as well.

Resolution: Subsumed by new issue **[Issue-0105: Types for nodes in the data model.]**.

Issue-0094: Static type errors and warnings

Date: 31-May-2001
Raised by: Don Chamberlin

Description: Static type errors and warnings are not specified. We need to enumerate in both the XQuery and formal semantics documents what kinds of static type errors and warnings are produced by the type system. See also **[Issue-0090: Static type-assertion expression]**.

Issue-0095: Importing Schemas and DTDs into query

Date: 31-May-2001
Raised by: Don Chamberlin

Description: We do not specify how a Schema or DTD is 'imported' into a query so that its information is available during type checking. Schema and DTDs can either be named explicitly (e.g., by an 'IMPORT SCHEMA' clause in a query) or implicitly, by accessing documents that refer to a Schema or DTD. The mechanism for statically accessing a Schema or DTD is unspecified.

Issue-0096: Support for schema-less and incompletely validated documents

Date: 31-May-2001
Raised by: Don Chamberlin/Mary Fernandez

Description: This is related to **[Issue-0095: Importing Schemas and DTDs into query]**. We do not specify what is the effect of type checking a query that is applied to a document without a DTD or Schema. In general, a schema-less document has type xs:AnyType and type checking can proceed under that assumption. A related issue is what is the effect of type checking a query that is applied to an incompletely validated document. As above, we can make *no* assumptions about the static type of an incompletely validated document and must assume its static type is xs:AnyType.

Issue-0097: Static type-checking vs. Schema validation

Date: 31-May-2001
Raised by: Mary Fernandez

Description: Static type checking and schema validation are not equivalent, but we might want to do both in a query. For example, we might want to assert statically that an expression has a particular type and also

validate dynamically the value of an expression w.r.t a particular schema.

The differences between static type checking and schema validation must be enumerated clearly (the XSFD people should help us with this).

Issue-0098: Implementation of and conformance levels for static type checking

Date: 31-May-2001
Raised by: Don Chamberlin

Description: This issue is related to **[Issue-0059: Testing Subtyping]** Static type checking may be difficult and/or expensive to implement. Some discussion of algorithmic issues of type checking are needed. In addition, we may want to define "conformance levels" for XQuery, in which some processors (or some processing modes) are more permissive about types. This would allow XQuery implementations that do not understand all of Schema, and it would allow customers some control over the cost/benefit tradeoff of type checking.

Issue-0099: Incomplete/inconsistent mapping from to core

Date: 06-June-2001
Raised by: Don Chamberlin

Description: This mapping is still preliminary and contains inconsistencies. These inconsistencies will be addressed in detail in the next draft of the document.

Resolution: The Formal Semantics now provides a complete mapping from XQuery to the Core XQuery. Remaining issues with respect to that mapping are indicated separately.

Issue-0100: Namespace resolution

Date: March-11-2002
Raised by: FS Editors

The way (when? where?) namespace prefixes are resolved is still an open issue.

Issue-0101: Support for mixed content in the type system

Date: March-11-2002
Raised by: FS Editors

Description: Support for mixed content in the type system is an open issue. This reopens issue **[Issue-0016: Mixed content]**. Dealing with mixed content with interleaving raises complexity issue. See also **[Issue-0103: Complexity of interleaving]**.

Issue-0102: Indentation, Whitespace

Date: March-11-2002
Raised by: FS Editors

Description: Whitespace normalization in XQuery is still an open issue. This reopens issue **[Issue-0022: Indentation, Whitespaces]**.

Issue-0103: Complexity of interleaving

Date: March-11-2002

Raised by: FS Editors

Description: The current type system allows interleaving is allowed on arbitrary types. Interleaving is an expensive operation and it is not clear how to define subtyping for it. Should we restrict use of interleaving on (optional) atomic types ? Should this restriction reflects the one in XML schema ? Related to **[Issue-0077: Interleaved repetition and closure]**.

Issue-0104: Support for named typing

Date: March-11-2002
Raised by: FS Editors

Description: XML Schema is based on named typing, while the XQuery type system is based on strucural typing. Directly related issues are **[Issue-0019: Support derived types]** and **[Issue-0018: Align algebra types with schema]**. Other impacted issues are **[Issue-0020: Structural vs. name equivalence]**, **[Issue-0072: Facet value access in Query Algebra]**, **[Issue-0073: Facets for simple types and their role for typechecking]**, **[Issue-0088: Align types with XML Schema : Formal Description.]**, **[Issue-0095: Importing Schemas and DTDs into query]**, **[Issue-0097: Static type-checking vs. Schema validation]**, **[Issue-0098: Implementation of and conformance levels for static type checking]**, **[Issue-0111: Semantics of instance of ... only]**.

Issue-0105: Types for nodes in the data model.

Date: March-11-2002
Raised by: FS Editors

Description: The XQuery type system only supports element and attribute nodes. It needs to support other kinds of nodes from the XQuery datamodel, notably: text nodes and document nodes. Should it also include support for PI nodes, comment nodes and namespace nodes?

The data model distinguishes between text nodes and strings, which are simple-typed values. Text nodes have identity, parents, and siblings. Strings do not. Text nodes are accessed by the children() accessor; strings and other simple-typed values are accessed by the typed-value() accessor. The distinction between text nodes and simple-typed values should exist in type system as well.

Issue-0106: Constraint on attribute and element content models

Date: March-11-2002
Raised by: Jerome

Description: The XQuery type system allows more content model than what XML Schema allows. For instance, the current type grammar allows the following types:

```
element d { (attribute a | element b, attribute c)* }
attribute a { element b }
```

Section **[3 The XQuery Type System]** indicates corresponding constraints on the XQuery type system to avoid that problem. The status of these constraints is unclear. When are they enforced and checked?

Issue-0107: Semantics of data()

Date: March-11-2002

Raised by: FS Editors

Description: What is the semantics of data() applied to anything else than an element or attribute node ?

Issue-0108: Principal node types in XPath

Date: March-11-2002
Raised by: Michael Kay

Description: There is a known bug in the formal semantics which does not deal properly with principal node types. This bug should be resolved based on some semantics previously proposed by Phil Wadler.

Issue-0109: Semantics of sortby

Date: March-11-2002
Raised by: Jerome

Description: The precise semantics of sortby is still open issue.

Issue-0110: Semantics of element and attribute constructors

Date: March-11-2002
Raised by: Jerome

Description: The precise semantics of element constructors is still an open issue.

Issue-0111: Semantics of instance of ... only

Date: March-11-2002
Raised by: Mary Fernandez

Description: The "instance of" expression allows an optional "only" modifier. The use case for such a modifier is based on named typing, while the XQuery semantics is currently based on structural typing. It is not clear what the semantics of the "only" modifier under structural typing should be and how it can be supported.

Issue-0112: Typing for the typeswitch default clause

Date: March-11-2002
Raised by: Jerome

Description: There is an asymetry in the typing for the default clause in typeswitch vs. the other case clauses. This results in a less precise type when the default clause can be applied.

It would be nicer to be able to have the type be more precise, like for the other case clauses.

The technical problem is the need for some form of negation. I think one could define a "non-common-primes" function that would do the trick, but I leave that as open for now until further review of the new typeswitch section is made.

Issue-0113: Incomplete specification of type conversions

Date: March-11-2002
Raised by: Mary Fernandez

Description: Not all the fallback conversion rules are specified yet in section **[4.1.2 Type Conversions]**. All

the remaining rules in the fallback conversions table must be specified.

Issue-0114: Dynamic context for current date and time

Date: March-11-2002
Raised by: Jerome

Description: The following components dynamic contexts have no formal representation yet: current date and time.

Related question: where are these context components used?

Issue-0115: What is in the default context?

Date: March-11-2002
Raised by: Jerome

Description: What do the default namespace and type environments contain? I believe at least the default namespace environment should contain the "xs", "xf" and "op" prefixes, as well as the default namespaces bound to the empty namespace. Should the default type environment contain wildcard types?

Issue-0116: Serialization

Date: March-11-2002
Raised by: Jerome

Description: Serialization of data model instances, and XQuery results is still an open issue.

Issue-0117: Data model constructor for error values

Date: March-11-2002
Raised by: Jerome

Description: The XQuery data model supports an error value, but there is no constructor for it. Currently the formal semantics is using the notation dm:error() to create an error value. Should there be some function(s) in the [XQuery 1.0 and XPath 2.0 Functions and Operators] document to create error values?

Issue-0118: Data model syntax and literal values

Date: March-11-2002
Raised by: Phil Wadler

Description: Phil suggests the data model should support primitive literals in their lexical form, in which case no explicit dynamic semantic rule would be necessary.

More generally, should the data model support a constructor syntax?

Issue-0119: Semantics of op:to

Date: March-11-2002
Raised by: Mary Fernandez

Description: The binary operator "to" is not defined on empty sequences. The [XQuery 1.0 and XPath 2.0 Functions and Operators] document says operands are decimals, while the XQuery document says they are integers. What happens when Expr1 > Expr2?

Issue-0120: Sequence operations: value vs. node identity

Date: March-11-2002
Raised by: Mary Fernandez

Description: The [XQuery 1.0 and XPath 2.0 Functions and Operators] document provides only one function for union (intersect, etc.) of sequences of nodes and values. The semantics is very different for node only and value only sequences. The semantics is undefined for heterogeneous sequences. Should we have two union (intersect, etc.) functions, one for nodes, and one for values?

Issue-0121: Casting functions

Date: March-11-2002
Raised by: Jerome

Description: The [XQuery 1.0 and XPath 2.0 Functions and Operators] document does not provide any function for casting, just a table and casting rules. Wouldn't it be preferable to either have an explicit function to normalize to? This relates to Issue 17 in the [XQuery 1.0 and XPath 2.0 Functions and Operators] document.

Issue-0122: Overloaded functions

Date: March-11-2002
Raised by: Denise Draper

Description: Some [XQuery 1.0 and XPath 2.0 Functions and Operators] functions are overloaded. How to deal with overloaded built-in functions in the Formal Semantics is still an open issue.

Issue-0123: Semantics of /

Date: March-11-2002
Raised by: FS Editors

Description: Some of the semantics of the root expression `/` is still an open issue. For instance, what should be the semantics of `'/'` in case of a document fragment (I.e., created using XQuery element constructor).

Issue-0124: Binding position in FLWR expressions

Date: March-11-2002
Raised by: FS Editors

Description: The only way to bind position() is through implicit operations. It would be useful and cleaner to also have a way to bind position in a sequence explicitely. The FS editors proposed several syntax for such an operation, one of these syntaxes look like "for $v at $i in E1 return E2" which modifies the for expression to bind a variable "$i" to the position in sequence "E1" explicitely.

Issue-0125: Operations on node only in XPath

Date: March-11-2002
Raised by: FS Editors

Description: Generally steps may operate on nodes and values alike; the axis rules only can operate on nodes (NodeValue). Is it a dynamic error to apply an axis rule on a value?

More generally, the XQuery document states that XPath operates on nodes only. Where that restriction should

be applied is an open issue.

Issue-0126: Semantics of effective boolean value

Date: March-11-2002
Raised by: FS Editors

Description: Some of the semantics of effective boolean value is still an open issue.

Issue-0127: Datatype limitations

Date: March-11-2002
Raised by: FS Editors

Description: Should the Datatype production allow the following: fs:atomic, fs:numeric, fs:UnknownSimpleType ?

The formal semantics makes use of several built-in types which are not in XML Schema: fs:numeric, fs:atomic, and fs:UnknownSimpleType. These types are necessary for the specification of some of XPath type conversion rules, and will be accepted without raising an error.

Issue-0128: Casting based on the lexical form

Date: March-11-2002
Raised by: Mary Fernandez

Description: The XQuery/XPath spec says : "If an operand is an untyped simple value (...), it is cast to the type suggested by its lexical form." It is not clear how to define the semantics for this.

Issue-0129: Static typing of union

Date: March-11-2002
Raised by: Michael Rys

Description: What should be the semantics of arithmetics expressions over unions. Right now, it would raise a dynamic error. Do we want to raise a static error?

Should operators and functions consistenly with respect to typing?

With the current semantics in Section 4.5 expr1 + expr2 raises a static type error if (e.g.) expr1 has type string and expr2 has type integer. It raises only a dynamic error, if expr1 has type (string | integer) and expr2 has type integer, and expr1 actually evaluates to a string. An alternative would be that this raises also a static error, because it cannot be guarantueed to succeed on all instances.

Issue-0130: When to process the query prolog

Date: March-11-2002
Raised by: Jerome

Description: The query prolog needs to be processed before the normalization phase. This is not reflected yet in the processing model.

Issue-0131: Boolean node test and sequences

Date: March-11-2002
Raised by: Michael Rys

Description: The current semantics for boolean node tests makes it "true if the expression is a sequence that contains at least one node and error if the sequence contains no node". This is inefficient to implement. Some alternative semantics have been proposed.

Issue-0132: Typing for descendant

Date: March-11-2002
Raised by: Peter Fankhauser

Description: The current static typing for descendant is still under review and the inferences rules in that version are probably containing bugs.

Issue-0133: Should to also be described in the formal semantics?

Date: March-11-2002
Raised by: Michael Kay

Description: The current semantics of the op operator is using the op:to function from the [XQuery 1.0 and XPath 2.0 Functions and Operators] document. Should it be defined formally?

Mike Kay suggests the following definition: `[A to B] => if A=B then (A) else (A, A+1 to B)`

Issue-0134: Should we define for with head and tail?

Date: March-11-2002
Raised by: Michael Kay

Description: Mike Kay proposes to use the following recursion to define the dynamic semantics of for:

```
[for $x in () return E2] => ()
[for $x in SEQ return E2] =>
(let $x := head(SEQ) return E2, for $x in tail(SEQ) return E2)
```

Unfortunately head and tail are not define in [XQuery 1.0 and XPath 2.0 Functions and Operators] right now.

Issue-0135: Semantics of special functions

Date: March-11-2002
Raised by: Michael Kay

Description: The current semantics does not completely cover built-in functions. Some functions used in the Formal semantics, or some functions from the [XQuery 1.0 and XPath 2.0 Functions and Operators] document need additional semantics specification.

C.3 Alphabetic list of issues

Open Issues

C.3.1 Open Issues

Number: 58

[Issue-0018: Align algebra types with schema]
[Issue-0088: Align types with XML Schema : Formal Description.]
[Issue-0124: Binding position in FLWR expressions]
[Issue-0131: Boolean node test and sequences]
[Issue-0128: Casting based on the lexical form]
[Issue-0121: Casting functions]
[Issue-0103: Complexity of interleaving]
[Issue-0106: Constraint on attribute and element content models]
[Issue-0117: Data model constructor for error values]
[Issue-0118: Data model syntax and literal values]
[Issue-0127: Datatype limitations]
[Issue-0068: Document Collections]
[Issue-0114: Dynamic context for current date and time]
[Issue-0073: Facets for simple types and their role for typechecking]
[Issue-0072: Facet value access in Query Algebra]
[Issue-0008: Fixed point operator or recursive functions]
[Issue-0078: Generation of ambiguous types]
[Issue-0098: Implementation of and conformance levels for static type checking]
[Issue-0095: Importing Schemas and DTDs into query]
[Issue-0113: Incomplete specification of type conversions]
[Issue-0102: Indentation, Whitespace]
[Issue-0033: Metadata Queries]
[Issue-0100: Namespace resolution]
[Issue-0125: Operations on node only in XPath]
[Issue-0122: Overloaded functions]
[Issue-0014: Polymorphic types]
[Issue-0108: Principal node types in XPath]
[Issue-0123: Semantics of /]
[Issue-0107: Semantics of data()]
[Issue-0126: Semantics of effective boolean value]
[Issue-0110: Semantics of element and attribute constructors]
[Issue-0111: Semantics of instance of ... only]
[Issue-0119: Semantics of op:to]
[Issue-0109: Semantics of sortby]
[Issue-0135: Semantics of special functions]
[Issue-0120: Sequence operations: value vs. node identity]
[Issue-0116: Serialization]
[Issue-0133: Should to also be described in the formal semantics?]
[Issue-0134: Should we define for with head and tail?]
[Issue-0097: Static type-checking vs. Schema validation]
[Issue-0094: Static type errors and warnings]
[Issue-0129: Static typing of union]
[Issue-0020: Structural vs. name equivalence]

[Issue-0019: Support derived types]
[Issue-0101: Support for mixed content in the type system]
[Issue-0104: Support for named typing]
[Issue-0096: Support for schema-less and incompletely validated documents]
[Issue-0089: Syntax for types in XQuery]
[Issue-0059: Testing Subtyping]
[Issue-0082: Type and expression operator precedence]
[Issue-0105: Types for nodes in the data model.]
[Issue-0055: Types with non-wellformed instances]
[Issue-0132: Typing for descendant]
[Issue-0112: Typing for the typeswitch default clause]
[Issue-0080: Typing of parent]
[Issue-0115: What is in the default context?]
[Issue-0130: When to process the query prolog]
[Issue-0011: XPath tumbler syntax instead of index?]

Resolved (or redundant) Issues

C.3.2 Resolved (or redundant) Issues

Number: 77

[Issue-0015: 3-valued logic to support NULLs]
[Issue-0071: Alignment with the XML Query Datamodel]
[Issue-0091: Attribute expression]
[Issue-0001: Attributes]
[Issue-0047: Attributes]
[Issue-0030: Automatic type coercion]
[Issue-0052: Axes of XPath]
[Issue-0065: Built-In GroupBy?]
[Issue-0027: Case syntax]
[Issue-0040: Case Syntax]
[Issue-0023: Catch exceptions and process in algebra?]
[Issue-0013: Collations]
[Issue-0010: Construct values by copy]
[Issue-0038: Copy by reachability]
[Issue-0037: Copy vs identity semantics]
[Issue-0039: Dereferencing semantics]
[Issue-0003: Document Order]
[Issue-0063: Do we need (user defined) higher order functions?]
[Issue-0058: Downward Navigation only?]
[Issue-0005: Element identity]
[Issue-0064: Error code handling in Query Algebra]
[Issue-0092: Error expression]

[**Issue-0035: Exception handling**]
[**Issue-0084: Execution model**]
[**Issue-0048: Explicit Type Declarations**]
[**Issue-0083: Expressive power and complexity of typeswitch expression**]
[**Issue-0009: Externally defined functions**]
[**Issue-0046: FOR Syntax**]
[**Issue-0032: Full regular path expressions**]
[**Issue-0028: Fusion**]
[**Issue-0034: Fusion**]
[**Issue-0045: Global Order**]
[**Issue-0036: Global-order based operators**]
[**Issue-0079: Global order between nodes in different documents**]
[**Issue-0054: Global vs. local complex types**]
[**Issue-0053: Global vs. local elements**]
[**Issue-0042: GroupBy**]
[**Issue-0012: GroupBy - needs second order functions?**]
[**Issue-0099: Incomplete/inconsistent mapping from to core**]
[**Issue-0022: Indentation, Whitespaces**]
[**Issue-0077: Interleaved repetition and closure**]
[**Issue-0060: Internationalization aspects for strings**]
[**Issue-0044: Keys and IDREF**]
[**Issue-0081: Lexical representation of Schema simple types**]
[**Issue-0016: Mixed content**]
[**Issue-0061: Model for References**]
[**Issue-0087: More examples of Joins**]
[**Issue-0057: More precise type system; choice in path**]
[**Issue-0002: Namespaces**]
[**Issue-0062: Open questions for constructing elements by reference**]
[**Issue-0074: Operational semantics for expressions**]
[**Issue-0056: Operators on Simple Types**]
[**Issue-0069: Organization of Document**]
[**Issue-0075: Overloading user defined functions**]
[**Issue-0026: Project - one tag only**]
[**Issue-0051: Project redundant?**]
[**Issue-0050: Recursive Descent for XPath**]
[**Issue-0043: Recursive Descent for XPath**]
[**Issue-0031: Recursive functions**]
[**Issue-0007: References: IDREFS, Keyrefs, Joins**]
[**Issue-0004: References vs containment**]
[**Issue-0093: Representation of Text Nodes in type system**]
[**Issue-0067: Runtime Casts**]
[**Issue-0085: Semantics of Wildcard type**]
[**Issue-0066: Shallow or Deep Equality?**]

[Issue-0041: Sorting]
[Issue-0006: Source and join syntax instead of "for"]
[Issue-0070: Stable vs. Unstable Sort/Distinct]
[Issue-0090: Static type-assertion expression]
[Issue-0086: Syntactic rules]
[Issue-0021: Syntax]
[Issue-0025: Treatment of empty results at type level]
[Issue-0049: Unordered Collections]
[Issue-0017: Unordered content]
[Issue-0076: Unordered types]
[Issue-0024: Value for empty sequences]
[Issue-0029: Views]

C.4 Delegated Issues

C.4.1 XPath 2.0

The following issues are delegated to XPath 2.0: **[Issue-0007: References: IDREFS, Keyrefs, Joins]**, **[Issue-0030: Automatic type coercion]**, **[Issue-0032: Full regular path expressions]**, **[Issue-0052: Axes of XPath]**.

C.4.2 XQuery

The following issues are delegated to XQuery: **[Issue-0009: Externally defined functions]**, **[Issue-0028: Fusion]**, **[Issue-0029: Views]**, **[Issue-0030: Automatic type coercion]**, **[Issue-0052: Axes of XPath]**, **[Issue-0081: Lexical representation of Schema simple types]**, **[Issue-0082: Type and expression operator precedence]**, **[Issue-0086: Syntactic rules]**.

C.4.3 Operators

The following issues are delegated to XPath 2.0:**[Issue-0013: Collations]**, **[Issue-0030: Automatic type coercion]**, **[Issue-0056: Operators on Simple Types]**, **[Issue-0060: Internationalization aspects for strings]**.