ECMAnScript for XML (E4X)
Specification

Final Draft – 10 February 2004
ECMAScript for XML (E4X)
Specification
Brief History

On 13 June 2002, a group of companies led by BEA Systems proposed a set of programming language extensions adding native XML support to ECMAScript (ECMA-262). The programming language extensions were designed to provide a simple, familiar, general purpose XML programming model that flattens the XML learning curve by leveraging the existing skills and knowledge of one of the largest developer communities worldwide. The benefits of this XML programming model include reduced code complexity, tighter revision cycles, faster time to market, decreased XML footprint requirements and looser coupling between code and XML data.

The ECMAScript group (ECMA TC39/TG1) unanimously agreed to the proposal and established a sub-group to standardize the syntax and semantics of a general purpose, cross platform, vendor neutral set of programming language extensions called ECMAScript for XML (E4X). The development of this Standard started on 8 August 2002. This Standard was developed as an extension to ECMAScript Edition 3, but may be applied to other versions of ECMAScript as well.

This Standard adds native XML datatypes to the ECMAScript language, extends the semantics of familiar ECMAScript operators for manipulating XML data and adds a small set of new operators for common XML operations, such as searching and filtering. It also adds support for XML literals, namespaces, qualified names and other mechanisms to facilitate XML processing.

This standard will be integrated into future editions of ECMA-262 (ECMAScript). The ECMAScript group is working on significant enhancements for future editions of the ECMAScript language, including mechanisms for defining XML types using the XML Schema language and support for classes.

This Standard has been adopted as the 1st Edition of ECMA-<TBD ###> by the ECMA General Assembly in June, 2004.

The following people have contributed to this specification:

- John Schneider, BEA/AgileDelta (Lead Editor)
- Jeff Dyer, Macromedia (Supporting Editor)
- Rok Yu, Microsoft (Supporting Editor)
- Steve Adamski, AOL/Netscape
- Patrick Beard, AOL/Netscape
- Adam Bosworth, BEA
- Steve Brandli, BEA
- Waldemar Horwat, AOL/Netscape
- Ethan Hugg, AgileDelta
- Mark Igra, BEA
- Alex Khesin, BEA
- Terry Lucas, BEA
- Richard Rollman, AgileDelta
- Markus Scherer, IBM
- Michael Shenfield, RIM
- Peter Torr, Microsoft
- Eric Vasilik, BEA
- Wayne Vicknair, IBM
- Herman Venter, Microsoft
- Roger Weber, BEA

This list is incomplete. If your name has been omitted, please let me know so I can add it. Also, please provide any additional people (internal or otherwise) who contributed to this specification by providing comments or feedback. Thanks!
Table of contents

1 Scope 1
2 Status of this Document 1
3 Normative References 1
4 Motivation 2
  4.1 The Rise of XML Processing 2
  4.2 Current XML Processing Approaches 2
    4.2.1 The Document Object Model (DOM) 2
    4.2.2 The eXtensible Stylesheet Language (XSLT) 2
    4.2.3 Object Mapping 2
  4.3 The E4X Approach 3
5 Design Principles 4
6 Notational Conventions 5
  6.1 Algorithm Conventions 5
    6.1.1 Indentation Style 5
    6.1.2 Property Access 5
    6.1.3 Iteration 7
    6.1.4 Conditional Repetition 7
  6.2 Method Invocation 8
7 Lexical Conventions 9
  7.1 Punctuators 10
  7.2 XML Initialiser Input Elements 10
8 Types 12
  8.1 The XML Type 12
    8.1.1 Internal Properties and Methods 12
      8.1.1.1 [[Get]] (P) 13
      8.1.1.2 [[Put]] (P, V) 13
      8.1.1.3 [[Delete]] (P) 15
      8.1.1.4 [[DefaultValue]] (hint) 16
      8.1.1.5 [[HasProperty]] (P) 16
      8.1.1.6 [[DeepCopy]] () 16
      8.1.1.7 [[Descendants]] (P) 17
8.1.1.8  [Evaluates] (V)  
8.1.1.9  [ResolveValue] ()  
8.1.1.10  [[Insert]] (P, V)  
8.1.1.11  [[Replace]] (P, V)  
8.1.1.12  [[AddInScopeNamespace]] (N)  
8.2  The XMLList Type  
8.2.1  Internal Properties and Methods  
8.2.1.1  [[Get]] (P)  
8.2.1.2  [[Put]] (P, V)  
8.2.1.3  [[Delete]] (P)  
8.2.1.4  [[DefaultValue]] (hint)  
8.2.1.5  [[HasProperty]] (P)  
8.2.1.6  [[Append]] (V)  
8.2.1.7  [[DeepCopy]] ()  
8.2.1.8  [[Descendants]] (P)  
8.2.1.9  [[Equals]] (V)  
8.2.1.10  [[ResolveValue]] ()  
8.3  The AttributeName Type  
8.3.1  Internal Properties  
8.4  The AnyName Type  

9  Type Conversion  
9.1  ToString  
9.1.1  ToString Applied to the XML Type  
9.1.2  ToString Applied to the XMLList Type  
9.2  ToXMLString (input argument, [AncestorNamespaces], [IndentLevel])  
9.2.1  ToXMLString Applied to the XML Type  
9.3  ToXML  
9.3.1  ToXML Applied to the String Type  
9.3.2  ToXML Applied to a W3C XML Information Item  
9.4  ToXMLList  
9.4.1  ToXMLList Applied to the String Type  
9.5  ToAttributeName  
9.5.1  ToAttributeName Applied to the String Type  
9.6  ToXMLName  
9.6.1  ToXMLName Applied to the String Type  

10  Expressions  
10.1  Primary Expressions  
10.1.1  Attribute Identifiers  
10.1.2  Qualified Identifiers  
10.1.3  Wildcard Identifiers  
10.1.4  XML Initialiser  
10.1.5  XMLList Initialiser  
10.2  Left-Hand-Side Expressions  
10.2.1  Property Accessors  
10.2.2  Function Calls
10.2.2.1 CallMethod ( r, args ) 47
10.2.3 XML Descendant Accessor 47
10.2.4 XML Filtering Predicate Operator 48
10.3 Unary Operators 49
10.3.1 The Delete Operator 49
10.3.2 The typeof Operator 50
10.4 Additive Operators 51
10.4.1 The Addition Operator (+) 51
10.5 Equality Operators 52
10.5.1 The Abstract Equality Comparison Algorithm 52
10.6 Assignment Operators 53
10.6.1 XML Assignment Operator 53
10.6.2 XMLList Assignment Operator 54
10.6.3 Compound Assignment (op=) 55

11 Statements 57
11.1 Default XML Namespace Statement 57
11.2 The for-in Statement 58

12 Native E4X Objects 61
12.1 The Global Object 61
12.1.1 Internal Properties of the Global Object 61
12.1.2 Function Properties of the Global Object 61
12.1.3 Constructor Properties of the Global Object 61
12.2 Namespace Objects 62
12.2.1 The Namespace Constructor Called as a Function 62
12.2.2 The Namespace Constructor 62
12.2.3 Properties of the Namespace Constructor 63
12.2.4 Properties of the Namespace Prototype Object (Built-in Methods) 63
12.2.5 Properties of Namespace Instances 64
12.3 QName Objects 64
12.3.1 The QName Constructor Called as a Function 64
12.3.2 The QName Constructor 64
12.3.3 Properties of the QName Constructor 65
12.3.4 Properties of the QName Prototype Object 65
12.3.5 Properties of QName Instances 66
12.4 XML Objects 67
12.4.1 The XML Constructor Called as a Function 67
12.4.2 The XML Constructor 68
12.4.3 Properties of the XML Constructor 68
12.4.3.7 XML.settings ( ) 69
12.4.3.8 XML.setSettings ( [ Settings ] ) 69
12.4.3.9 XML.defaultSettings ( ) 70
12.4.4 Properties of the XML Prototype Object (Built-in Methods) 70
12.4.5 Properties of XML Instances 82
12.5 XMLList Objects 82
12.5.1 The XMLList Constructor Called as a Function 82
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.5.2</td>
<td>The XMLList Constructor</td>
<td>83</td>
</tr>
<tr>
<td>12.5.3</td>
<td>Properties of the XMLList Constructor</td>
<td>83</td>
</tr>
<tr>
<td>12.5.3.1</td>
<td>XMLList.prototype</td>
<td>83</td>
</tr>
<tr>
<td>12.5.4</td>
<td>Properties of the XMLList Prototype Object (Built-in Methods)</td>
<td>83</td>
</tr>
<tr>
<td>13</td>
<td>Errors</td>
<td>90</td>
</tr>
<tr>
<td>A</td>
<td>Optional Features (Non-normative)</td>
<td>91</td>
</tr>
<tr>
<td>A.1</td>
<td>XML Built-in Methods</td>
<td>91</td>
</tr>
<tr>
<td>A.2</td>
<td>XMLList Built-in Methods</td>
<td>91</td>
</tr>
</tbody>
</table>
1 Scope
This standard defines the syntax and semantics of ECMAScript for XML (E4X), a set of programming language extensions adding native XML support to ECMAScript.

2 Status of this Document
This document is a final draft produced to motivate, facilitate and finalize discussions related to E4X with the goal of creating a general purpose, cross platform, vendor neutral E4X standard. It is considered stable, but has not yet been agreed or otherwise sanctioned by ECMA. Comments and suggestions are solicited and encouraged.

3 Normative References
Document Object Model (DOM) Level 2 Specifications, W3C Recommendation, 13 November 2000
Extensible Markup Language 1.0 (Second Edition), W3C Recommendation 6 October 2000
Namespaces in XML, W3C Recommendation, 14 January 1999
ISO/IEC 10646-1:1993 Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus its amendments and corrigenda.
XML Information Set, W3C Recommendation 24 October 2001
XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999
4 Motivation

4.1 The Rise of XML Processing

Developing software to create, navigate and manipulate XML data is a significant part of every Internet developer’s job. Developers are inundated with data encoded in the eXtensible Markup Language (XML). Web pages are increasingly encoded using XML vocabularies, including XHTML and Scalable Vector Graphics (SVG). On mobile devices, data is encoded using the Wireless Markup Language (WML). Web services interact using the Simple Object Access Protocol (SOAP) and are described using the Web Service Description Language (WSDL). Deployment descriptors, project make files and configuration files and now encoded in XML, not to mention an endless list of custom XML vocabularies designed for vertical industries. XML data itself is even described and processed using XML in the form of XML Schemas and XSL Stylesheets.

4.2 Current XML Processing Approaches

Current XML processing techniques require ECMAScript programmers to learn and master a complex array of new concepts and programming techniques. The XML programming models often seem heavyweight, complex and unfamiliar for ECMAScript programmers. This section provides a brief overview of the more popular XML processing techniques.

4.2.1 The Document Object Model (DOM)

One of the most common approaches to processing XML is to use a software package that implements the interfaces defined by the W3C XML DOM (Document Object Model). The XML DOM represents XML data using a general purpose tree abstraction and provides a tree-based API for navigating and manipulating the data (e.g., getParentNode(), getChildNodes(), removeChild(), etc.).

This method of accessing and manipulating data structures is very different from the methods used to access and manipulate native ECMAScript data structures. ECMAScript programmers must learn to write tree navigation algorithms instead of object navigation algorithms. In addition, they have to learn a relatively complex interface hierarchy for interacting with the XML DOM. The resulting XML DOM code is generally harder to read, write, and maintain than code that manipulates native ECMAScript data structures. It is more verbose and often obscures the developer’s intent with lengthy tree navigation logic. Consequently, XML DOM programs require more time, knowledge and resources to develop.

4.2.2 The eXtensible Stylesheet Language (XSLT)

XSLT is a language for transforming XML documents into other XML documents. Like the XML DOM, it represents XML data using a tree-based abstraction, but also provides an expression language called XPath designed for navigating trees. On top of this, it adds a declarative, rule-based language for matching portions of the input document and generating the output document accordingly.

From this description, it is clear that XSLT’s methods for accessing and manipulating data structures are completely different from those used to access and manipulate ECMAScript data structures. Consequently, the XSLT learning curve for ECMAScript programmers is quite steep. In addition to learning a new data model, ECMAScript programmers have to learn a declarative programming model, recursive decent processing model, new expression language, new XML language syntax, and a variety of new programming concepts (templates, patterns, priority rules, etc.). These differences also make XSLT code harder to read, write and maintain for the ECMAScript programmer. In addition, it is not possible to use familiar development environments, debuggers and testing tools with XSLT.

4.2.3 Object Mapping

Several have also tried to navigate and manipulate XML data by mapping it to and from native ECMAScript objects. The idea is to map XML data onto a set of ECMAScript objects, manipulate those objects directly, then map them back to XML. This allows ECMAScript programmers to reuse their knowledge of ECMAScript objects to manipulate XML data.

This is a great idea, but unfortunately it does not work for a wide range of XML processing tasks. Native ECMAScript objects do not preserve the order of the original XML data and order is significant for XML. Not only do XML developers need to preserve the order of XML data, but they also need to control and manipulate the order of XML data. In addition, XML data contains artifacts that are not easily represented by the ECMAScript object model, such as namespaces, attributes, comments, processing instructions and mixed element content.
4.3 The E4X Approach

ECMAScript for XML was envisioned to address these problems. E4X extends the ECMAScript object model with native support for XML data. It reuses familiar ECMAScript operators for creating, navigating and manipulating XML, such that anyone who has used ECMAScript is able to start using XML with little or no additional knowledge. The extensions include native XML data types, XML literals (i.e., initialisers) and a small set of new operators useful for common XML operations, such as searching and filtering.

E4X applications are smaller and more intuitive to ECMAScript developers than comparable XSLT or DOM applications. They are easier to read, write and maintain requiring less developer time, skill and specialized knowledge. The net result is reduced code complexity, tighter revision cycles and shorter time to market for Internet applications. In addition, E4X is a lighter weight technology enabling a wide range of mobile applications.
5 Design Principles

The following design principles are used to guide the development of E4X and encourage consistent design decisions. They are listed here to provide insight into the E4X design rational and to anchor discussions on desirable E4X traits.

- **Simple**: One of the most important objectives of E4X is to simplify common programming tasks. Simplicity should not be compromised for interesting or unique features that do not address common programming problems.

- **Consistent**: The design of E4X should be internally consistent such that developers can anticipate its behaviour.

- **Familiar**: Common operators available for manipulating ECMAScript objects should also be available for manipulating XML data. The semantics of the operators should not be surprising to those familiar with ECMAScript objects. Developers already familiar with ECMAScript objects should be able to begin using XML objects with minimal surprises.

- **Minimal**: Where appropriate, E4X defines new operators for manipulating XML that are not currently available for manipulating ECMAScript objects. This set of operators should be kept to a minimum to avoid unnecessary complexity. It is a non-goal of E4X to provide, for example, the full functionality of XPath.

- **Loose Coupling**: To the degree practical, E4X operators will enable applications to minimize their dependencies on external data formats. For example, E4X applications should be able to extract a value deeply nested within an XML structure, without specifying the full path to the data. Thus, changes in the containment hierarchy of the data will not require changes to the application.

- **Complementary**: E4X should integrate well with other languages designed for manipulating XML, such as XPath, XSLT and XML Query. For example, E4X should be able to invoke complementary languages when additional expressive power is needed without compromising the simplicity of the E4X language itself.
6 Notational Conventions

This specification extends the notational conventions used in the ECMAScript Edition 3 specification. In particular, it extends the algorithm notation to improve the clarity, readability and maintainability of this specification. The new algorithm conventions are described in this section.

6.1 Algorithm Conventions

This section introduces the algorithm conventions this specification adds to those used to describe the semantics of ECMAScript Edition 3. These conventions are not part of the E4X language. They are used within this specification to describe the semantics of E4X operations.

6.1.1 Indentation Style

This specification extends the notation used in the ECMAScript Edition 3 specification by defining an algorithm indentation style. The new algorithm indentation style is used in this specification to group related collections of steps together. This convention is useful for expressing a set of steps that are taken conditionally or repeatedly. For example, the following algorithm fragment uses indentation to describe a set of steps that are taken conditionally.

1. If resetParameters is true
   a. Let \( x = 0 \)
   b. Let \( y = 0 \)
   c. Let \( \Delta X = 0.5 \)
2. Else
   a. Let \( \Delta X = \Delta X + \text{accelerationX} \)

In the example above, steps 1.a through 1.c are taken if the condition expressed in step 1 evaluates to true. Otherwise, step 2.a is taken.

Standard outline numbering form is used to identify steps and distinguish nested levels of indentation when it might not otherwise be obvious due to pagination.

6.1.2 Property Access

This specification extends the notation used in the ECMAScript Edition 3 specification by defining three property access conventions. When used on the left hand side of an assignment operation in this specification, the property access conventions are used to modify the value of a specified property of a specified object. In other contexts in this specification, the property access conventions are used for specifying that the value of a specified property be retrieved from a specified object based on its property name.

There are three forms of the property access conventions, two for accessing normal properties and one for accessing internal properties. The first convention for accessing normal properties is expressed using the following notation:

\[
\text{object}.\text{propertyName}
\]

When used on the left hand side of an assignment operation, this property access convention is equivalent to calling the \([\text{Put}]\) method of \(\text{object}\), passing the string literal containing the same sequence of parameters as \(\text{propertyName}\) and the value from the right hand side of the assignment operator as arguments. For example, the following algorithm fragment:

1. Let \( \text{item}\!.\text{price} = "5.95" \)

is equivalent to the following algorithm fragment:

1. Call the \([\text{Put}]\) method of \(\text{item}\) with arguments "price" and "5.95"

When used in other contexts, this property access convention is equivalent to calling the \([\text{Get}]\) method of \(\text{object}\) passing the string literal containing the same sequence of characters as \(\text{propertyName}\) as an argument and returning the result. For example, the following algorithm fragment:

1. Let \( \text{price} = \text{item}\!.\text{price} \)
is equivalent to the following algorithm fragment:

1. Let \( \text{price} \) be the result of calling the \([\text{Get}]\) method of \( \text{item} \) with argument "price"

The second convention for accessing normal properties is expressed using the following notation:

\[
\text{object}[\text{propertyName}]
\]

When used on the left hand side of an assignment operation, this property access convention is equivalent to calling the \([\text{Put}]\) method of \( \text{object} \) passing \( \text{ToString(propertyName)} \) and the value from the right hand side of the assignment operator as arguments. For example, the following algorithm fragment:

1. Let \( \text{item}[1] = \text{item2} \)

is equivalent to the following algorithm fragment:

1. Call the \([\text{Put}]\) method of \( \text{item} \) with arguments \( \text{ToString(1)} \) and \( \text{item2} \)

When used in other contexts, this property access convention is equivalent to calling the \([\text{Get}]\) method of \( \text{object} \) with argument \( \text{ToString(propertyName)} \) and returning the result. For example, the following algorithm fragment:

1. Let \( \text{item2} = \text{item}[1] \)

is equivalent to the following algorithm fragment:

1. Let \( \text{item2} \) be the result of calling the \([\text{Get}]\) method on \( \text{item} \) with argument \( \text{ToString(1)} \)

This is a convenient and familiar notation for specifying numeric property names used as array indices. It is also used for setting and retrieving the value of properties based on a computed property name.

The convention for accessing internal property names, including those that refer to internal methods, is specified using the following notation:

\[
\text{object}.[[\text{internalPropertyName}]]
\]

When used on the left hand side of an assignment operation, this property access convention is equivalent to setting the value of the \([\text{internalPropertyName}]\) of the specified \( \text{object} \) to the value from the right hand side of the assignment operator. For example, the following algorithm fragment:

1. Let \( x.[[\text{Class}]] = \text{"element"} \)

is equivalent to the following algorithm fragment:

1. Let the value of the \([\text{Class}]\) property of \( x \) be \text{"element"}

When used in other contexts, this property access convention is equivalent to getting the value of of the \([\text{internalPropertyName}]\) property of \( \text{object} \). For example, the following algorithm fragment:

1. Let \( \text{class} = x.[[\text{Class}]] \)

is equivalent to the following algorithm fragment:

1. Let \( \text{class} \) be the value of the \([\text{Class}]\) property of \( x \)
6.1.3 Iteration

This specification extends the notation used for describing ECMAScript Edition 3 by defining two iteration conventions. These iteration conventions are used by this specification for expressing that a set of steps must be taken once for each item in a collection or once for each integer in a specified range.

The first iteration convention is defined for expressing a sequence of steps that must be taken once for each member of a collection. It is expressed using the following for each notation:

For each item in collection steps

This for each notation is equivalent to performing the given steps repeatedly with the variable item bound to each member of collection. The value of collection is computed once prior to performing steps and does not change while performing steps. The order in which item is bound to members of collection is implementation dependent. The repetition ends after item has been bound to all the members of collection or when the algorithm exits via a return or a thrown exception. The steps may be specified on the same line following a comma or on the following lines using the indentation style described in section 6.1.1. For example,

1. Let total = 0
2. For each product in groceryList
   a. If product.price > maxPrice, throw an exception
   b. Let total = total + product.price

In this example, steps 2.a and 2.b are repeated once for each member of the collection groceryList. The variable product is bound to each member of groceryList before each repetition of these steps.

The second iteration convention defined by this specification is for expressing a sequence of steps that must be repeated once for each integer in a specified range of integers. It is expressed using the following for notation:

For variable = first to last steps

This for notation is equivalent to computing first and last, which will evaluate to integers i and j respectively, and performing the given steps repeatedly with the variable variable bound to each member of the sequence i, i+1 … j in numerical order. The repetition ends after variable has been bound to each item of this sequence or when the algorithm exits via a return or a thrown exception. If i is greater than j, the steps are not performed. The steps may be specified on the same line following a comma or on the following lines using the indentation style described above. For example,

1. For i = 0 to priceList.length-1, call ToString(priceList[i])

In this example, ToString is called once for each item in priceList in sequential order.

A modified version of the for notation exists for iterating through a range of integers in reverse sequential order. It is expressed using the following notation:

For variable = first downto last steps

The modified for notation works exactly as described above except the variable variable is bound to each member of the sequence i, i-1,.. j in reverse numerical order. If i is less than j, the steps are not performed.

6.1.4 Conditional Repetition

This specification extends the notation used in the ECMAScript Edition 3 specification by defining a convention for expressing conditional repetition of a set of steps. This convention is defined by the following notation:

While ( expression ) steps

The while notation is equivalent to computing the expression, which will evaluate to either true or false and if it is true, taking the given steps and repeating this process until the expression evaluates to false or the algorithm exits via a return or a thrown exception. The steps may be specified on the same line following a comma or on the following lines using the indentation style described above. For example,
1. Let $\log_2 = 0$
2. While ($n > 1$)
   a. Let $n = n / 2$
   b. Let $\log_2 = \log_2 + 1$

In this example, steps 2.a and 2.b are repeated until the expression $n > 1$ evaluates to \textbf{false}.

### 6.2 Method Invocation

This specification extends the notation used in the ECMAScript Edition 3 specification by defining a method invocation convention. The method invocation convention is used in this specification for calling a method of a given object passing a given set of arguments and returning the result. This convention is defined by the following notation:

$$\textit{object} \ . \ \textit{methodName} ( \ \textit{arguments} )$$

where \textit{arguments} is a comma separated list of zero or more values. The method invocation notation is equivalent to constructing a new Reference \textit{r} with \textit{base object} set to \textit{object} and \textit{property name} set to a string literal containing the same sequence of characters as \textit{methodName}, constructing an internal list \textit{l} of the values in \textit{arguments}, invoking the CallMethod operator (section 10.2.2.1) passing \textit{r} and \textit{l} as arguments and returning the result. For example, the following algorithm fragment:

1. Let \textit{sub} = \textit{s}.substring(2, 5)

Is equivalent to the following algorithm fragment:

1. Let \textit{r} be a new Reference with \textit{base object} = \textit{s} and \textit{property name} = "substring"
2. Let \textit{l} be an internal list containing the values 2 and 5
3. Let \textit{sub} = CallMethod(\textit{r}, \textit{l})
7 Lexical Conventions

This section introduces the lexical conventions E4X adds to ECMAScript.

E4X modifies the existing lexical grammar productions for InputElementRegExp and Punctuators. It also introduces the goal symbols InputElementXMLTag and InputElementXMLContent that describe how sequences of Unicode characters are translated into parts of XML initialisers.

In addition to the grammar contexts where a division (/) or division-assignment (/=) is permitted, the InputElementDiv symbol is used in those syntactic grammar contexts where a less than (<), less than or equals (<=), left shift (<<) or left shift-assignment (<<=) operator is permitted. The InputElementXMLTag is used in those syntactic contexts where the literal contents of an XML tag are permitted. The InputElementXMLContent is used in those syntactic contexts where the literal contents of an XML element are permitted. The InputElementRegExp symbol is used in all other syntactic grammar contexts.

The addition of the production InputElementRegExp :: XMLMarkup and extended use of the existing production InputElementRegExp :: Punctuator :: < allow the start of XML initialisers to be identified.

To better understand when these goal symbols apply, consider the following example:

```plaintext
order = <{x}>{item}</{x}>;
```

The input elements returned from the lexical grammar along with the goal symbol and productions used for this example are as follows:

<table>
<thead>
<tr>
<th>Input Element</th>
<th>Goal</th>
<th>Productions</th>
</tr>
</thead>
<tbody>
<tr>
<td>order</td>
<td>InputElementRegExp</td>
<td>Token :: Identifier</td>
</tr>
<tr>
<td>=</td>
<td>InputElementDiv</td>
<td>Punctuator</td>
</tr>
<tr>
<td>&lt;</td>
<td>InputElementRegExp</td>
<td>Punctuator</td>
</tr>
<tr>
<td>{</td>
<td>InputElementXMLTag</td>
<td>{</td>
</tr>
<tr>
<td>x</td>
<td>InputElementRegExp</td>
<td>Token :: Identifier</td>
</tr>
<tr>
<td>}</td>
<td>InputElementDiv</td>
<td>Punctuator</td>
</tr>
<tr>
<td>&gt;</td>
<td>InputElementXMLTag</td>
<td>XMLTagPunctuator</td>
</tr>
<tr>
<td>}</td>
<td>InputElementXMLContent</td>
<td>{</td>
</tr>
<tr>
<td>item</td>
<td>InputElementRegExp</td>
<td>Token :: Identifier</td>
</tr>
<tr>
<td>}</td>
<td>InputElementDiv</td>
<td>Punctuator</td>
</tr>
<tr>
<td>&lt;/</td>
<td>InputElementXMLContent</td>
<td>&lt;/</td>
</tr>
<tr>
<td>{</td>
<td>InputElementXMLTag</td>
<td>{</td>
</tr>
<tr>
<td>x</td>
<td>InputElementRegExp</td>
<td>Token :: Identifier</td>
</tr>
<tr>
<td>}</td>
<td>InputElementDiv</td>
<td>Punctuator</td>
</tr>
<tr>
<td>&gt;</td>
<td>InputElementXMLTag</td>
<td>XMLTagPunctuator</td>
</tr>
<tr>
<td>;</td>
<td>InputElementRegExp</td>
<td>Token :: Punctuator</td>
</tr>
</tbody>
</table>

Syntax

E4X extends the InputElementRegExp goal symbol defined by ECMAScript with the following production:

```
InputElementRegExp :: XMLMarkup
```

E4X extends ECMAScript by adding the following goal symbols:

```
InputElementXMLTag ::
  XMLTagCharacters
  XMLTagPunctuator
  XMLAttributeValue
  {
```
InputElementXMLContent ::
    XMLMarkup
    XMLText
    {
    <
    </

7.1 Punctuators

E4X adds the descendent (..) input element to Punctuator to support the XML descendent accessor (section 10.2.3) with the following production:

    Punctuator ::
        ..

7.2 XML Initialiser Input Elements

The goal symbols InputElementXMLTag and InputElementXMLContent describe how Unicode characters are translated into input elements that describe parts of XML initialisers. These input elements are consumed by the syntactic grammars described in sections 10.1.4 and 10.1.5.

The lexical grammar allows characters which may not form a valid XML initialiser. The syntax and semantics described in the syntactic grammar ensure that the final initialiser is well formed XML.

Unlike in string literals, the back slash (\) is not treated as the start of an escape sequence inside XML initialisers. Instead the XML entity references specified in the XML 1.0 specification should be used to escape characters. For example, the entity reference &apos; can be used for a single quote ('), &quot; for a double quote ("), and &lt; for less than (<).

The left curly brace ({) and right curly brace (}) are used to delimit expressions that may be embedded in tags or element content to dynamically compute portions of the XML initialiser. The curly braces may appear in literal form inside an attribute value, a CDATA, PI, or XML Comment. In all other cases, the character reference &#x7B; must be used to represent the left curly brace ({) and the character reference &#x7D; must be used to represent the right curly brace (}).

Syntax

XMLMarkup ::
    XMLComment
    XMLCDATA
    XMLPI

XMLElementCharacters ::
    SourceCharacters but no embedded XMLTagPunctuator
    or left-curly { or quote ' or double-quote "

XMLText ::
    SourceCharacters but no embedded left-curly { or less-than <

XMLComment ::
    <!-- XMLCommentCharacters_opt -->

XMLCommentCharacters ::
    SourceCharacters but no embedded sequence --

XMLCDATA ::
    <![CDATA[ XMLCDATACharacters_opt ]]>

XMLCDATACharacters ::
    SourceCharacters but no embedded sequence ]]>
<? XMLPICharacters_opt ?>

XMLPICharacters ::
    SourceCharacters but no embedded sequence ?>

XMLAttributeValue::
    " XMLDoubleStringCharacters_opt "
    ' XMLSingleStringCharacters_opt '

XMLDoubleStringCharacters ::
    SourceCharacters but no embedded double-quote "

XMLSingleStringCharacters ::
    SourceCharacters but no embedded single-quote '

SourceCharacters ::
    SourceCharacter SourceCharacters_opt

XMLTagPunctuator :: one of
    = > /

/>
8 Types
E4X extends ECMAScript by adding two new fundamental data types for representing XML elements and XML lists. Future versions will also provide the capability to derive user-defined types for specific XML vocabularies using XML Schemas.

8.1 The XML Type
The XML type is an ordered collection of properties with a name, a set of XML attributes, a set of in-scope namespaces and a parent. Each property of an XML object has a unique numeric property name \( P \), such that ToString(ToUint32(\( P \))) is equal to \( P \), and has a value of type XML representing a child node. The name of an XML object is a QName object or null. Each XML attribute is an instance of the XML type. Each namespace is a Namespace object. The parent is a value of type XML or null. Methods are associated with XML objects using non-numeric property names.

Each value of type XML represents an XML element, attribute, comment, processing-instruction or text node. The internal [[Class]] property is set to “element”, “attribute”, “comment”, “processing-instruction” or “text” as appropriate. Each XML object representing an XML attribute, comment, processing-instruction (PI) or text node has no properties and stores a String value representing the value of the associated attribute, comment, PI or text node in the [[Value]] property logically inherited from the Object type.

8.1.1 Internal Properties and Methods
Internal properties and methods are not part of the E4X language. They are defined by this specification purely for expository purposes. An implementation of E4X must behave as if it produced and operated upon internal properties in the manner described here. This specification reuses the notation for internal properties from the ECMAScript Edition 3 specification, wherein the names of internal properties are enclosed in double square brackets [[ ]]. When an algorithm uses an internal property of an object and the object does not implement the indicated internal property, a TypeError exception is thrown.

The XML type is logically derived from the Object type and inherits its internal properties. The following table summarises the internal properties the XML type adds to those defined by the Object type.

<table>
<thead>
<tr>
<th>Property</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Name]]</td>
<td>None</td>
<td>The name of this XML object.</td>
</tr>
<tr>
<td>[[Parent]]</td>
<td>None</td>
<td>The parent of this XML object.</td>
</tr>
<tr>
<td>[[Attributes]]</td>
<td>None</td>
<td>The attributes associated with this XML object.</td>
</tr>
<tr>
<td>[[InScopeNamespaces]]</td>
<td>None</td>
<td>The namespaces in scope for this XML object</td>
</tr>
<tr>
<td>[[Length]]</td>
<td>None</td>
<td>The number of ordered properties in this XML object.</td>
</tr>
<tr>
<td>[[DeepCopy]]</td>
<td>()</td>
<td>Returns a deep copy of this XML object.</td>
</tr>
<tr>
<td>[[ResolveValue]]</td>
<td>()</td>
<td>Returns this XML object. This method is used when attempting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to resolve the value of an empty XMLList.</td>
</tr>
<tr>
<td>[[Descendants]]</td>
<td>(PropertyName)</td>
<td>Returns an XMLList containing the descendants of this XML</td>
</tr>
<tr>
<td></td>
<td></td>
<td>object with names that match propertyName.</td>
</tr>
<tr>
<td>[[Equals]]</td>
<td>(Value)</td>
<td>Returns a Boolean value indicating whether this XML object has</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the same XML content as the given XML Value.</td>
</tr>
<tr>
<td>[[Insert]]</td>
<td>(PropertyName, Value)</td>
<td>Inserts one or more new properties before the property with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>name propertyName (a numeric index).</td>
</tr>
<tr>
<td>[[Replace]]</td>
<td>(PropertyName, Value)</td>
<td>Replaces the value of the property with name propertyName (a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>numeric index) with one or more new properties</td>
</tr>
<tr>
<td>[[AddInScopeNamespace]]</td>
<td>(Namespace)</td>
<td>Adds the given Namespace to the [[InScopeNamespaces]] property</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of this XML object.</td>
</tr>
</tbody>
</table>

The value of the [[Name]] property must be null or a QName object containing a legal XML element name, attribute name, or PI name. The value of the [[Name]] property is null if and only if the XML object represents an XML comment or text node. The [[Name]] for each XML object representing a processing-instruction will have its uri property set to the empty string.

The value of the [[Parent]] property must be either an XML object or null. When an XML object occurs as a property (i.e., a child) of another XML object, the [[Parent]] property is set to the containing XML object (i.e., the parent).

The value of the [[Attributes]] property is a set of zero or more XML objects. When a new object is added to the [[Attributes]] set, it replaces any existing object in [[Attributes]] that has the same set identity. The set identity of each XML object \( x \in \)
[[Attributes]] is defined to be \(x.[[[Name]]]\). Therefore, there exists no two objects \(x, y \in [[Attributes]]\) such that the result of the comparison \(x.[[[Name]]] = y.[[[Name]]]\) is true. The value of the \([[Attributes]]\) property is the empty set if the XML object represents an XML attribute, comment, PI or text node. Note: Although namespaces are declared using attribute syntax in XML, they are not represented in the \([[Attributes]]\) property.

The value of the \([[InScopeNamespaces]]\) property is a set of zero or more Namespace objects representing the namespace declarations in scope for this XML object. When a new object is added to the \([[InScopeNamespaces]]\) set, it replaces any existing object in the \([[InScopeNamespaces]]\) set that has the same set identity. The set identity of each Namespace object \(n \in [[InScopeNamespaces]]\) is defined to be \(n\).prefix. Therefore, there exists no two objects \(x, y \in [[InScopeNamespaces]]\), such that the result of the comparison \(x.prefix = y.prefix\) is true.

The value of the \([[Length]]\) property is a positive integer.

### 8.1.1.1 [[Get]] (P)

**Overview**

The XML type overrides the internal [[Get]] method defined by the Object type. The XML [[Get]] method is used to retrieve the value of a property by its numeric property name, an XML attribute by its name or a set of XML elements by their name. The input variable \(P\) may be a numeric property name, an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading “@” symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard “*” or the attributes wildcard “@*”. When the input variable \(P\) is an unqualified XML element name, it identifies XML elements in the default namespace. When the input variable \(P\) is an unqualified XML attribute name, it identifies XML attributes in no namespace.

Note: Unlike the internal Object.[[Get]] method, the internal XML [[Get]] method is never used for retrieving methods associated with XML objects. E4X modifies the ECMAScript method lookup semantics for XML objects as described in section 10.2.2.

**Semantics**

When the [[Get]] method of an XML object \(x\) is called with property name \(P\), the following steps are taken:

1. If `ToString(ToUint32(P)) == P`
   a. Call the Object.[[Get]] method with \(x\) as the this object and argument \(P\), then return the result
2. Let \(n\) = ToXMLName(P)
3. Let \(l\) be a new XMLList with \(l[[TargetObject]] = x\) and \(l[[TargetProperty]] = n\)
4. If Type(n) is AttributeName
   a. For each \(a\) in \(x.[[[Attributes]]]\)
      i. If \((n.[[[Name]], localName == "*")) or (n.[[[Name]], localName == a.[[[Name]], localName])
         and (n.[[[Name]], uri == undefined) or (n.[[[Name]], uri == a.[[[Name]], uri])
         1. Call the [[Append]] method of \(l\) with argument \(a\)
   b. Return \(l\)
5. For \((k = 0\) to \(x.[[[Length]]]-1)\)
   a. If \((n.localName == "*"))
      or ((\(x[k].[[Class]] == \"element\") and \((x[k].[[Name]], localName == n.localName)))
      and ((n.uri == undefined) or (n.uri == x[k].[[Name]].uri))
      i. Call the [[Append]] method of \(l\) with argument \(x[k]\)

### 8.1.1.2 [[Put]] (P, V)

**Overview**

The XML type overrides the internal [[Put]] method defined by the Object type. The XML [[Put]] method is used to replace and insert properties or XML attributes in an XML object. The parameter \(P\) identifies which portion of the XML object will be affected and may be a numeric property name, an unqualified name for an XML attribute (distinguished from XML valued property names by a leading “@” symbol) or set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes or the properties wildcard “*”. When the parameter \(P\) is an unqualified XML element name, it identifies XML elements in the default namespace. When the parameter \(P\) is an unqualified XML attribute name, it identifies
When the \texttt{Put} method of an XML object converted to a String with \texttt{ToString()}, \texttt{XML attributes}

\texttt{Semantics}.

When the \texttt{[\texttt{Put}]} method of an XML object \texttt{x} is called with property name \texttt{P} and value \texttt{V}, the following steps are taken:

1. If \texttt{x.[[\texttt{Class}]]} \in \{"text", "comment", "processing-instruction", "attribute"\}, return
2. If \texttt{(Type(V) \notin \{XML, XMLList\}) or (V.hasSimpleContent() \equiv true)}
   a. Let \texttt{c = ToString(V)}
3. Else
   a. Let \texttt{c} be the result of calling the \texttt{[\texttt{DeepCopy}]} method of \texttt{V}
4. If \texttt{ToString(ToUint32(P)) \equiv P}
   a. Call the \texttt{[\texttt{Replace}]} method of \texttt{x} with arguments \texttt{P} and \texttt{c} and return
5. Let \texttt{n = ToXMLName(P)}
6. Let \texttt{defaultNamespace = GetDefaultNamespace()}
7. If \texttt{Type(n) \equiv AttributeName}
   a. Call the function \texttt{isXMLName} (section 12.1.2.1) with argument \texttt{n.[[Name]]} and if the result is \texttt{false}, return
   b. If \texttt{Type(c) \equiv XMLList}
      i. If \texttt{c.[[Length]]} \equiv 0, let \texttt{c} be the empty string
      ii. Else
          1. Let \texttt{s = ToString(c[0])}
          2. For \texttt{i = 1 to c.[[Length]].1}
              a. Let \texttt{s be the result of concatenating s, the string " " (space) and ToString(c[i])}
          3. Let \texttt{c = s}
   c. Else
      i. Let \texttt{c = ToString(c)}
5. Let \texttt{a = null}
6. For \texttt{j = 0 to x.[[Attributes]].length}
   i. If \texttt{(n.[[Name]].localName \equiv x.[[Attributes]].j.[[Name]].localName)
      and (n.[[Name]].uri \equiv undefined) or (n.[[Name]].uri \equiv x.[[Attributes]].j.[[Name]].uri))}
       1. If \texttt{(a \equiv null), a = x.[[Attributes]].j}
 6. Else call the [[Delete]] method of \texttt{x} with argument \texttt{x.[[Attributes]].j.[[Name]]}
7. If \texttt{a \equiv null}
   i. Let \texttt{name be a new QName created as if by calling the constructor new QName(n.[[Name]])}
   ii. If \texttt{name.uri \equiv undefined}
      1. Let \texttt{nons be a new Namespace created as if by calling the constructor new Namespace()}
      2. Call the [[SetNamespace]] method of \texttt{name with argument nons}
   iii. Create a new XML object \texttt{a with a.[[Name]] = name, a.[[Class]] = "attribute" and a.[[Parent]] = x}
   iv. Let \texttt{x.[[XMLAttributes]] = x.[[XMLAttributes]] \cup a}
   v. Let \texttt{ns be the result of calling the [[GetNamespace]] method of name with no arguments}
   vi. Call the [[AddInScopeNamespace]] method of \texttt{x with argument ns}
   g. Let \texttt{a.[[Value]] = c}
   h. Return
8. Call the function \texttt{isXMLName} (section 12.1.2.1) with argument \texttt{n} and if the result is \texttt{false}, return
9. Let \texttt{i = undefined}
10. Let \texttt{primitiveAssign = (Type(c) \notin \{XML, XMLList\}) and (n.localName is not equal to the string "**")}
11. For \texttt{(k = 0 to x.[[Length]].1)}
    a. If \texttt{(n.localName \equiv "*"})
      or ((\texttt{x[k].[[Class]] \equiv "element"}) and (\texttt{x[k].[[Name]].localName==n.localName}))
      and ((\texttt{n.uri \equiv undefined}) or (\texttt{n.uri \equiv x[k].[[Name]].uri}))
      i. If \texttt{(i \equiv undefined), let i = k}
    ii. Else call the [[Delete]] method of \texttt{x with argument ToString(k)}
12. If \texttt{i \equiv undefined}
    a. Let \texttt{i = x.[[Length]]}
    b. Let \texttt{name be a new QName created as if by calling the constructor new QName(n)}
    c. If \texttt{primitiveAssign \equiv true}
i. If name.uri == undefined, call [[SetNamespace]] on name with argument defaultNamespace
ii. Create a new XML object y with y.[[Name]] = name, y.[[Class]] = "element" and y.[[Parent]] = x
iii. Let ns be the result of calling [[GetNamespace]] on name with no arguments
iv. Call the [[Replace]] method of x with arguments ToString(i) and y
v. Call [[AddInScopeNamespace]] on y with argument ns

13. If (primitiveAssign == true)
   a. Delete all the properties of the XML object x[i]
   b. Let s = ToString(c)
   c. If s is not the empty string, call the [[Replace]] method of x[i] with arguments "0" and s
14. Else
   a. Call the [[Replace]] method of x with arguments ToString(i) and c
15. Return

8.1.1.3 [[Delete]] (P)

Overview

The XML type overrides the internal [[Delete]] method defined by the Object type. The XML [[Delete]] method is used to remove a property by its numeric property name, as set of XML attributes by name or a set of XML valued properties by name. Unlike, the internal Object [[Delete]], the XML [[Delete]] method shifts all the properties following deleted properties up to fill in empty slots created by the delete. The input variable P may be a numeric property name, an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading ‘@’ symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard ‘*’ or the attributes wildcard ‘**’. When the input variable P is an unqualified XML element name, it identifies XML elements in the default namespace. When the input variable P is an unqualified XML attribute name, it identifies XML attributes in no namespace.

Semantics

When the [[Delete]] method of an XML object x is called with property name P, the following steps are taken:

1. Let i = ToUint32(P)
2. If ToString(i) == P
   a. If i is greater than or equal to x.[[Length]], return true
   b. Else
      i. Let x[P].[[Parent]] = null
      ii. Remove the property with the name P from x
      iii. For each property q of x such that ToUint32(q) > i, rename q to ToString(ToUint32(q) - 1)
      iv. Let x.[[Length]] = x.[[Length]] - 1
   c. Return true
3. Let n = ToXMLName(P)
4. If Type(n) is AttributeName
   a. For each a in x.[[Attributes]]
      i. If ((n.[[Name]].localName == "*")) or (n.localName == a.[[Name]].localName) and
         ((n.[[Name]].uri == undefined) or (n.[[Name]].uri == a.[[Name]].uri))
         1. Let a.[[Parent]] = null
         2. Remove the attribute a from x.[[Attributes]]
   b. Return true
5. Let dp = 0
6. For each property q in x
   a. If (n.localName == "*" or (x[q].[[Class]] == "element" and x[q].[[Name]].localName == n.localName)) and
      (n.uri == undefined) or (n.uri == x[q].[[Name]].uri)
      i. Let q.[[Parent]] = null
      ii. Remove the property q from x
      iii. Let dp = dp + 1
   b. Else
      i. If dp > 0, rename q to ToString(ToUint32(q) - dp)
7. Let x.[[Length]] = x.[[Length]] - dp
8. Return true.
8.1.1.4 [[DefaultValue]] (hint)

Overview

The XML type overrides the internal [[DefaultValue]] method defined by the Object type. The XML [[DefaultValue]] method returns a primitive value representing this XML object. Unlike, the [[DefaultValue]] method defined by the Object type, the XML [[DefaultValue]] method always returns a string. The hint parameter is ignored.

Semantics

When the [[DefaultValue]] method of an XML object x is called with parameter hint, the following step is taken:

1. Return ToString(x)

8.1.1.5 [[HasProperty]] (P)

Overview

The XML type overrides the internal [[HasProperty]] method defined by the Object type. The XML [[HasProperty]] method is used to determine whether this XML object contains an XML element or attribute by its name or ordinal position. The input variable P may be a numeric property name, an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading "@" symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard "*" or the attributes wildcard "@*". When the input variable P is an unqualified XML element name, it identifies XML elements in the default namespace. When the input variable P is an unqualified XML attribute name, it identifies XML attributes in no namespace.

Semantics

When the [[HasProperty]] method of an XML object x is called with property name P, the following steps are taken:

2. If ToString(ToUint32(P)) == P
   a. Return (ToUint32(P) < x.[[Length]])
3. Let n = ToXMLName(P)
4. If Type(n) is AttributeName
   a. For each a in x.[[Attributes]]
      i. If ((n.[[Name]].localName == "*") or (n.[[Name]].localName == a.[[Name]].localName))
         and (n.[[Name]].uri == undefined) or (n.[[Name]].uri == a.[[Name]].uri))
         1. Return true
   b. Return false
5. For (k = 0 to x.[[Length]]-1)
   a. If ((n.localName == "*"))
      or ((s[k].[Class] == "element") and (s[k].[Name].localName == n.localName)))
      and ((n.uri == undefined) or (n.uri == s[k].[Name].uri))
      i. Return true
6. Return false

8.1.1.6 [[DeepCopy]] ()

Overview

The XML type adds the internal [[DeepCopy]] method to the internal properties defined by the Object type. The XML [[DeepCopy]] method is used to create and return a deep copy of this object, including its attributes, properties, namespaces and the attributes, properties and namespaces of all its descendants. The internal [[Parent]] property of the return value is set to null and the internal [[Parent]] property of each copied descendant is reset to its newly copied parent as appropriate.

Semantics

When the [[DeepCopy]] method of an XML object x is called, the following steps are taken:

1. Let y be a new XML object
2. For each internal property \([p]\) of \(x\), except \([\text{Parent}]\), \([\text{Attributes}]\) and \([\text{InScopeNamespaces}]\)
   a. Create a copy \(c\) of \(x.\[p]\) and let the value of \(y.\[p]\) be \(c\)
3. For each \(ns \in x.\[\text{InScopeNamespaces}]\)
   a. Let \(ns2\) be a new Namespace created as if by calling the constructor new Namespace(ns.prefix, ns.uri)
   b. Let \(y.\[\text{InScopeNamespaces}]\) = \(y.\[\text{InScopeNamespaces}]\) \(\cup ns2\)
4. Let \(y.\[\text{Parent}]\) = null
5. For each attribute \(a \in x.\[\text{Attributes}]\)
   a. Let \(b\) be the result of calling the \([\text{DeepCopy}]\) method of \(a\)
   b. Let \(b.\[\text{Parent}]\) = \(y\)
   c. Let \(y.\[\text{Attributes}]\) = \(y.\[\text{Attributes}]\) \(\cup b\)
6. For \(i = 0\) to \(x.\[\text{Length}]\)-1
   a. Let \(c\) be the result of calling the \([\text{DeepCopy}]\) method of \(x[i]\)
   b. Let \(y[i].\[\text{Parent}]\) = \(y\)
   c. Let \(y[i].\[\text{Parent}]\) = \(y\)
7. Return \(y\)

8.1.1.7 \([\text{Descendants}]\) (P)

**Overview**

The XML type adds the internal \([\text{Descendants}]\) method to the internal properties defined by the Object type. The XML \([\text{Descendants}]\) method is used to retrieve all the XML valued descendants of this XML object (i.e., children, grandchildren, great-grandchildren, etc.) with names matching the input variable \(P\). The input variable \(P\) may be a numeric property name, an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading “@” symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard “*” or the attributes wildcard “@*”. When the input variable \(P\) is an unqualified XML element name, it identifies XML elements in the default namespace. When the input variable \(P\) is an unqualified XML attribute name, it identifies XML attributes in no namespace.

**Semantics**

When the \([\text{Descendants}]\) method of an XML object \(x\) is called with property name \(P\), the following steps are taken:

1. Let \(n = \text{ToXMLName}(P)\)
2. Let \(l\) be a new XMLList with \(l.\[\text{TargetObject}]\) = null
3. If \(\text{Type}(n)\) is AttributeName
   a. For each attribute \(a\) in \(x.\[\text{Attributes}]\)
      i. If \((n.\[\text{Name}]\).localName = "*"\) or \((n.\text{localName} = a.\[\text{Name}]\).localName\)
         and \((n.\[\text{Name}]\).uri = undefined\) or \((n.\[\text{Name}]\).uri = a.\[\text{Name}]\).uri\)
            1. Call the \([\text{Append}]\) method of \(l\) with argument \(a\)
   b. For \((k = 0\) to \(x.\[\text{Length}]\)-1\)
      a. If \((n.\text{localName} = "*"\)
         or \((x[k].\[\text{Class}] = "element"\) and \((x[k].\[\text{Name}]\).localName = n.\text{localName}\)
         and \((n.\text{uri} = undefined\) or \(n.\text{uri} = x[k].\[\text{Name}]\).uri\))
            i. Call the \([\text{Append}]\) method of \(l\) with argument \(x[k]\)
      b. Let \(dq\) be the result of calling the \([\text{Descendants}]\) method of \(x[k]\) with argument \(P\)
      c. If \(dq.\[\text{Length}]\) > 0, call the \([\text{Append}]\) method of \(l\) with argument \(dq\)
4. Return \(l\)

8.1.1.8 \([\text{Equals}]\) (V)

**Overview**

The XML type adds the internal \([\text{Equals}]\) method to the internal properties defined by the Object type. The XML \([\text{Equals}]\) method is used to compare this XML object for XML content equality with another XML object \(V\). The \([\text{Equals}]\) operator returns true if \(V\) is a value of type XML considered equal to this XML object. Otherwise, it returns false.

**Semantics**

When the \([\text{Equals}]\) method of an XML object \(x\) is called with value \(V\), the following steps are taken:
1. If Type(V) is not XML, return false
2. If x.[[Class]] is not equal to V.[[Class]], return false
3. if x.[[Name]] is not null
   a. If x.[[Name]].localName is not equal to V.[[Name]].localName, return false
   b. If x.[[Name]].uri is not equal to V.[[Name]].uri, return false
4. If x.[[Attributes]] does not contain the same number of items as V.[[Attributes]], return false
5. If x.[[Length]] is not equal to V.[[Length]], return false
6. If x.[[Value]] is not equal to V.[[Value]], return false
7. For each attribute a in x.[[Attributes]]
   a. If V.[[Attributes]] does not contain an attribute b, such that b.[[Name]].localName == a.[[Name]].localName, b.[[Name]].uri == a.[[Name]].uri  and b.[[Value]] == a.[[Value]], return false
8. For i = 0 to x.[[Length]]-1
   a. Let r be the result of calling the [[Equals]] method of x[i] with argument V[i]
   b. If r == false, return false
9. Return true

8.1.1.9 [[ResolveValue]] ( )

Overview
The XML type adds the internal [[ResolveValue]] method to the internal properties defined by the Object type. The XML [[ResolveValue]] method returns this XML object. It is used by the XMLList [[ResolveValue]] method to support [[Put]] operations on empty XMLLists.

Semantics
When the [[ResolveValue]] method of an XML object x is called, the following step is taken:

1. Return x

8.1.1.10 [[Insert]] (P, V)

Overview
The XML type adds the internal [[Insert]] method to the internal properties defined by the Object type. The XML [[Insert]] method is used to insert a value V at a specific position P. The input variable P must be a numeric property name. The input variable V may be a value of type XML, XMLList or any value that can be converted to a String with ToString().

Semantics
When the [[Insert]] method of an XML object x is called with property name P and value V, the following steps are taken:

2. If x.[[Class]] ∈ {"text", "comment", "processing-instruction", "attribute"}, return
3. Let i = ToUint32(P)
4. If (ToString(i) is not equal to P) throw a TypeError exception
5. Let n = 1
6. If Type(V) is XMLList, let n = V.[[Length]]
7. If n == 0, Return
8. For j = x.[[Length]]-1 downto i, rename property ToString(j) of x to ToString(i + n)
9. Let x.[[Length]] = x.[[Length]] + n
10. If Type(V) is XMLList
    a. For j = 0 to V.[[Length]-1], call the [[Replace]] method of x with arguments ToString(i + j) and V[j]
11. Else
    a. Call the [[Replace]] method of x with arguments i and V
12. Return

8.1.1.11 [[Replace]] (P, V)

Overview
The XML type adds the internal [[Replace]] method to the internal properties defined by the Object type. The XML [[Replace]] method may be used to replace the property at a specific position \( P \) with the value \( V \). The input variable \( P \) must be a numeric property name. The input variable \( V \) may be a value of type XML, XMLList or any value that can be converted to a string with ToString().

**Semantics**

When the [[Replace]] method of an XML object \( x \) is called with property name \( P \) and value \( V \), the following steps are taken:

1. If \( x.\\{Class\\} \in \{\\"text\\", \"comment\\", \"processing-instruction\\", \"attribute\\\} \), return
2. Let \( i = \text{ToUint32}(P) \)
3. If (ToText\((i)\) is not equal to \( P \)) throw a **TypeError** exception
4. If \( i \) is greater than or equal to \( x.\\{Length\\} \),
   a. Let \( P = \text{ToText}(x.\\{Length\\}) \)
   b. Let \( x.\\{Length\\} = x.\\{Length\\} + 1 \)
5. If Type\((V)\) is XML and \( V.\\{Class\\} \in \{\\"element\\", \"comment\\", \"processing-instruction\\", \"text\\\} \)
   a. Let \( V.\\{Parent\\} = x \)
   b. Let \( x[P].\\{Parent\\} = \text{null} \)
   c. Let the value of property \( P \) of \( x \) be \( V \)
       Note: The E4X data model does not enforce the constraint: \( \forall x \in \text{XML} : x.\\{InScopeNamespaces\\} \supseteq x.\\{Parent\\}.\\{InScopeNamespaces\\} \). However, implementations may at this point add namespaces from \( V.\\{InScopeNamespaces\\} \) to \( x \) or any ancestors of \( x \). Likewise, implementations may at this point add namespaces from \( x \) to \( V \) or any descendents of \( V \).
6. Else if Type\((V)\) is XMLList
   a. Call the [[Delete]] method of \( x \) with argument \( P \)
   b. Call the [[Insert]] method of \( x \) with arguments \( P \) and \( V \)
7. Else
   a. Let \( s = \text{ToText}(V) \)
   b. Create a new XML object \( t \) with \( t.\\{Class\\} = \"text\\", t.\\{Parent\\} = x \) and \( t.\\{Value\\} = s \)
   c. Let \( x[P].\\{Parent\\} = \text{null} \)
   d. Let the value of property \( P \) of \( x \) be \( t \)
8. Return

### 8.1.1.12 [[AddInScopeNamespace]] ( N )

**Overview**

The XML type adds the internal [[AddInScopeNamespace]] method to the internal properties defined by the Object type. The XML [[AddInScopeNamespace]] method is used to add a new Namespace to the [[InScopeNamespaces]] of a given XML object. The input variable \( N \) is a value of type Namespace to be added to the [[InScopeNamespaces]] property of this XML object.

**Semantics**

When the [[AddInScopeNamespace]] method of an XML object \( x \) is called with a namespace \( N \), the following steps are taken:

1. If \( x.\\{Class\\} \in \{\\"text\\", \"comment\\", \"processing-instruction\\", \"attribute\\\} \), return
2. If \( N.\\{prefix\\} = \text{undefined} \)
   a. Let \( \text{match} = \text{null} \)
   b. For each \( ns \) in \( x.\\{InScopeNamespaces\\} \)
      i. If \( ns.\\{uri\\} = N.\\{uri\\} \), let \( \text{match} = ns \)
   c. If \( \text{match} = \text{null} \),
      i. Let \( x.\\{InScopeNamespaces\\} = x.\\{InScopeNamespaces\\} \cup N \)
3. Else
   a. Let \( \text{match} = \text{null} \)
   b. For each \( ns \) in \( x.\\{InScopeNamespaces\\} \)
      i. If \( N.\\{prefix\\} = ns.\\{prefix\\} \), let \( \text{match} = ns \)
   c. If \( \text{match} \) is not \( \text{null} \) and \( \text{match.\\{uri\\} \text{ is not equal to } N.\\{uri\\} \), let \( \text{match.\\{prefix\\} = \text{undefined} \)
4. Note: The E4X data model does not enforce the constraint: \( \forall x \in \text{XML} : x.\{\text{InScopeNamespaces}\} \supseteq x.\{\text{InScopeNamespaces}\} \cup N \). However, implementations may at this point add \( N \) to the \( \{\text{InScopeNamespaces}\} \) property of any ancestors or descendants of \( x \).

5. Return

8.2 The XMLList Type

The XMLList type is an ordered collection of properties. Each property of an XMLList object has a unique numeric property name \( P \), such that \( \text{ToString(ToUint32}(P)) \) is equal to \( P \) and a value of type XML. Methods are associated with XMLList objects using non-numeric property names.

A value of type XMLList represents an XML document, XML fragment or an arbitrary collection of XML objects (e.g., a query result).

8.2.1 Internal Properties and Methods

The XMLList type is logically derived from the Object type and inherits its internal properties. The following table summarises the internal properties the XMLList type adds to those defined by the Object type.

<table>
<thead>
<tr>
<th>Property</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{Length}</td>
<td>None</td>
<td>The number of properties contained in this XMLList object.</td>
</tr>
<tr>
<td>{TargetObject}</td>
<td>None</td>
<td>The XML or XMLList object associated with this object that will be affected when items are inserted into this XMLList.</td>
</tr>
<tr>
<td>{TargetProperty}</td>
<td>None</td>
<td>The name of a property that may be created in the {TargetObject} when objects are added to an empty XMLList.</td>
</tr>
<tr>
<td>{Append}</td>
<td>(Value)</td>
<td>Appends a new property to the end of this XMLList object.</td>
</tr>
<tr>
<td>{DeepCopy}</td>
<td>()</td>
<td>Returns a deep copy of this XMLList object.</td>
</tr>
<tr>
<td>{Descendants}</td>
<td>(PropertyName)</td>
<td>Returns an XMLList containing all the descendants of values of in this XMLList that have names matching propertyName.</td>
</tr>
<tr>
<td>{Equals}</td>
<td>(Value)</td>
<td>Returns a Boolean value indicating whether this XMLList object has the same content as the given Value or this XMLList object contains an object that compares equal to the given Value.</td>
</tr>
<tr>
<td>{ResolveValue}</td>
<td>()</td>
<td>Resolves the value of this XML object. If this XML object is not empty, it is returned. Otherwise, {ResolveValue} attempts to create an appropriate value.</td>
</tr>
</tbody>
</table>

The value of the \{Length\} property is a non-negative Number.

8.2.1.1 \{Get\} (P)

Overview

The XMLList type overrides the internal \{Get\} method defined by the Object type. The XMLList \{Get\} method is used to retrieve a specific property of this XMLList object by its numeric property name or to iterate over the XML valued properties of this XMLList object retrieving their XML attributes by name or their XML valued properties by name. The input variable \( P \) may be a numeric property name, an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading “@” symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard “*” or the attributes wildcard “@*”. When the input variable \( P \) is an unqualified XML element name, it identifies XML elements in the default namespace. When the input variable \( P \) is an unqualified XML attribute name, it identifies XML attributes in no namespace.

Note: Unlike the internal Object.\{Get\} method, the internal XMLList \{Get\} method is never used for retrieving methods associated with XMLList objects. E4X modifies the ECMAScript method lookup semantics for XMLList objects as described in section 10.2.2.

Semantics
When the [[Get]] method of an XMLList object \( x \) is called with property name \( P \), the following steps are taken:

1. If \( \text{ToString}(\text{ToUint32}(P)) = P \)
   a. Return the result of calling the Object.[[Get]] method with \( x \) as the this object and argument \( P \)
2. Let \( l \) be a new XMLList with \( l.\text{[TargetObject]} = x \) and \( l.\text{[TargetProperty]} = P \)
3. For \( i = 0 \) to \( x.\text{[Length]}-1 \)
   a. If \( x[i].\text{[Class]} = "element" \)
      i. Let \( gg \) be the result of calling the [[Get]] method of \( x[i] \) with argument \( P \)
      ii. If \( gg.\text{[Length]} > 0 \), call the [[Append]] method of \( l \) with argument \( gg \)
4. Return \( l \)

### 8.2.1.2 [[Put]] (P, V)

#### Overview

The XMLList type overrides the internal [[Put]] method defined by the Object type. The XMLList [[Put]] method is used to modify or replace an XML object within the XMLList and the context of its parent. In addition, when the XMLList contains a single property with an XML object, the [[Put]] method is used to modify, replace, and insert properties or XML attributes of that value by name. The input variable \( P \) identifies which portion of the XMLList and associated XML objects will be affected and may be a numeric property name, an unqualified name for an XML attribute (distinguished from XML valued property names by a leading "@@" symbol) or set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes or the properties wildcard ";*". When the input variable \( P \) is an unqualified XML element name, it identifies XML elements in the default namespace. When the input variable \( P \) is an unqualified XML attribute name, it identifies XML attributes in no namespace. The input variable \( V \) may be a value of type XML, XMLList or any value that can be converted to a String with ToString().

#### Semantics

When the [[Put]] method of an XMLList object \( x \) is called with property name \( P \) and value \( V \), the following steps are taken:

1. Let \( i = \text{ToUint32}(P) \)
2. If \( \text{ToString}(i) = P \)
   a. If \( x.\text{[TargetObject]} \) is not \( \text{null} \)
      i. Let \( r \) be the result of calling the [[ResolveValue]] method of \( x.\text{[TargetObject]} \)
      ii. If \( r = \text{null} \), return
   b. Else let \( r = \text{null} \)
   c. If \( i \) is greater than or equal to \( x.\text{[Length]} \)
      i. If Type(\( r \)) is XMLList
         1. If \( r.\text{[Length]} \) is not equal to \( 1 \), return
         2. Else let \( r = r[0] \)
      ii. Create a new XML object \( y \) with \( y.\text{[Parent]} = r \), \( y.\text{[Name]} = x.\text{[TargetProperty]} \), \( y.\text{[Attributes]} = \{ \} \), \( y.\text{[Length]} = 0 \)
      iii. If Type(\( x.\text{[TargetProperty]} \)) is AttributeName
          1. Let attributeExists be the result of calling the [[Get]] method of \( r \) with argument \( y.\text{[Name]} \)
          2. if (attributeExists.\[Length\] > 0), return
          3. let \( y.\text{[Class]} = \text{"attribute"} \)
      iv. Else if \( x.\text{[TargetProperty]} \) is null or \( x.\text{[TargetProperty]}\).localName is ";*"
          1. Let \( y.\text{[Name]} = \text{null} \)
          2. Let \( y.\text{[Class]} = \text{"text"} \)
      v. Else let \( y.\text{[Class]} = \text{"element"} \)
   vi. Let \( i = x.\text{[Length]} \)
   vii. If \( y.\text{[Parent]} \) is not \( \text{null} \) and \( y.\text{[Class]} \) is not equal to ";attribute"
      1. If \( i > 0 \)
         a. Let \( j = 0 \)
         b. While \( i < y.\text{[Parent]}\).\[Length\]-1 \)
            and \( y.\text{[Parent]}[j] \) is not the same object as \( x[j-1] \)
            i. Let \( j = j + 1 \)
      2. Else
a. Let \( i = y.([\text{Parent}], [\text{Length}]) - 1 \)

3. Call the \([[\text{Insert}]\]] method of \( y.([\text{Parent}]) \) with arguments ToString\((j+1)\) and \( y \)

viii. Call the \([[\text{Append}]\]] method of \( x \) with argument \( y \)

d. If \( \text{Type}(V) \notin \{\text{XML}, \text{XMLList}\} \) or \( (V.\text{hasSimpleContent}()== \text{true}) \), let \( V = \text{ToString}(V) \)

e. If \( x[i].[\{\text{Class}\}] == "\text{attribute}" \)

i. Call the \([[\text{Put}]\]] method of \( x[i].[\{\text{Parent}\}] \) with arguments \( x[i].[\{\text{Name}\}] \) and \( V \)

ii. Let \( \text{attr} \) be the result of calling \([[\text{Get}]\]] on \( x[i].[\{\text{Parent}\}] \) with argument \( x[i].[\{\text{Name}\}] \)

iii. Let \( x[i] = \text{attr}[0] \)

f. Else if \( \text{Type}(V) \) is \( \text{XMLList} \)

i. Create a shallow copy \( c \) of \( V \)

ii. Let \( \text{parent} = x[i].[\{\text{Parent}\}] \)

iii. If \( \text{parent} \) is not \( \text{null} \)

1. Let \( q \) be the property of \( \text{parent} \), such that \( \text{parent}[q] \) is the same object as \( x[i] \)

2. Call the \([[\text{Put}]\]] method of \( \text{parent} \) with arguments \( q \) and \( c \)

3. For \( j = 0 \) to \( c.([\text{Length}]) - 1 \)

\[ \text{Let } c[j] = \text{parent}[\text{ToString}(q + j)] \]

iv. For \( j = x.([\text{Length}]) - 1 \) down to \( i \), rename property \( j \) of \( x \) to \( \text{toString}(j + c.([\text{Length}])) \)

v. For \( j = 0 \) to \( c.([\text{Length}]) - 1 \), let \( x[i + j] = c[j] \)

vi. Let \( x.([\text{Length}]) = x.([\text{Length}]) + c.([\text{Length}]) \)

g. Else if \( \text{Type}(V) \) is \( \text{XML} \) or \( (x[i].[\{\text{Class}\}] \in \{"\text{text}" , "\text{comment}" , "\text{processing-instruction}"\}) \)

i. Let \( \text{parent} = x[i].[\{\text{Parent}\}] \)

ii. If \( \text{parent} \) is not \( \text{null} \)

1. Let \( q \) be the property of \( \text{parent} \), such that \( \text{parent}[q] \) is the same object as \( x[i] \)

2. Call the \([[\text{Put}]\]] method of \( \text{parent} \) with arguments \( q \) and \( V \)

3. Let \( V = \text{parent}[q] \)

iii. Let \( x[i] = V \)

h. Else

i. Call the \([[\text{Put}]\]] method of \( x[i] \) with arguments "**" and \( V \)

3. Else if \( x.([\text{Length}]) \) is less than or equal to \( 1 \)

a. If \( x.([\text{Length}]) == 0 \)

i. Let \( r \) be the result of calling the \([[\text{ResolveValue}]\]] method of \( x \)

ii. If \( (r == \text{null}) \) or \( (r.([\text{Length}]) \) is not equal to \( 1 \), return

iii. Call the \([[\text{Append}]\]] method of \( x \) with argument \( r \)

b. Call the \([[\text{Put}]\]] method of \( x[0] \) with arguments \( P \) and \( V \)

4. Return

### 8.2.1.3 \([[\text{Delete}]\]] (P)

#### Overview

The XMLList type overrides the internal \([[\text{Delete}]\]] method defined by the Object type. The XMLList \([[\text{Delete}]\]] method is used to remove a specific property of the XMLList by its numeric property name or to iterate over the XML valued properties of the XMLList removing their XML attributes or elements by name. The input variable \( P \) may be a numeric property name, an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading “@” symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard “**” or the attributes wildcard “@**”. When the input variable \( P \) is an unqualified XML element name, it identifies XML elements in the default namespace. When the input variable \( P \) is an unqualified XML attribute name, it identifies XML attributes in no namespace.

#### Semantics

When the \([[\text{Delete}]\]] method of an XMLList object \( x \) is called with property name \( P \), the following steps are taken:

1. Let \( i = \text{ToInt32}(P) \)

2. If \( \text{ToString}(i) == P \)

a. If \( i \) is greater than or equal to \( x.([\text{Length}]) \), return \( \text{true} \)

b. Else

i. Let \( \text{parent} = x[i].[\{\text{Parent}\}] \)

ii. If \( \text{parent} \) is not \( \text{null} \)

...
1. If \( x[i].[[\text{Class}]] == \text{"attribute"} \)
   a. Call the [[Delete]] method of parent with argument \( x[i].[[\text{Name}]] \)
2. Else
   a. Let \( q \) be the property of parent, where parent[\( q \)] is the same object as \( x[i] \)
   b. Call the [[Delete]] method of parent with argument \( q \)
   iii. Remove the property with the name \( P \) from \( x \)
   iv. For each property \( q \) in \( x \) such that ToUint32(\( q \)) \( > i \), rename \( q \) to Tostring(ToUint32(\( q \)) \( - 1 \))
   v. Let \( x.[[\text{Length}]] = x.[[\text{Length}]] - 1 \)

3. For each property \( q \) in \( x \),
   a. If \( x[q].[[\text{Class}]] == \text{"element"} \)
      i. Call the [[Delete]] method of \( x[q] \) with argument \( P \)
4. Return true

8.2.1.4 [[DefaultValue]] (\( hint \))

Overview

The XMLList type overrides the internal [[DefaultValue]] method defined by the Object type. The XMLList [[DefaultValue]] method returns a primitive value representing this XMLList object. Unlike the Object [[DefaultValue]] method, the XMLList [[DefaultValue]] method always returns a string. The hint parameter is ignored.

Semantics

When the [[DefaultValue]] method of an XMLList object \( l \) is called with parameter \( hint \), the following step is taken:

1. Return ToString(\( l \))

8.2.1.5 [[HasProperty]] (\( P \))

Overview

The XMLList type overrides the internal [[HasProperty]] method defined by the Object type. The XMLList [[HasProperty]] method is used to determine whether this XMLList object contains an XML element or attribute by its ordinal position or whether any of the objects contained in this XMLList object contains an XML element or attribute by its name. The input variable \( P \) may be a numeric property name, an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading “@” symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard “**” or the attributes wildcard “@*”. When the input variable \( P \) is an unqualified XML element name, it identifies XML elements in the default namespace. When the input variable \( P \) is an unqualified XML attribute name, it identifies XML attributes in no namespace.

Semantics

When the [[HasProperty]] method of an XMLList object \( x \) is called with property name \( P \), the following steps are taken:

1. If ToString(ToUint32(\( P \))) == \( P \)
   a. Return (ToUint32(\( P \)) \( < x.[[\text{Length}]] \))
2. For \( i = 0 \) to \( x.[[\text{Length}]] - 1 \)
   a. If \( x[i].[[\text{Class}]] == \text{"element"} \) and the result of calling the [[HasProperty]] method of \( x[i] \) with argument \( P \) == true, return true
3. Return false

8.2.1.6 [[Append]] (\( V \))

Overview

The XMLList type adds the internal [[Append]] method to the internal properties defined by the Object type. The XMLList [[Append]] method is used to append zero or more values specified by \( V \) to the end of the XMLList. The input variable \( V \) may be a value of type XMLList or XML.

Semantics
When the `[[Append]]` method of an XMLList object `x` is called with value `V`, the following steps are taken:

1. Let `i = x.[[Length]]`
2. Let `n = 1`
3. If `Type(V)` is XMLList,
   a. Let `x.[[TargetObject]] = V.[[TargetObject]]`
   b. Let `x.[[TargetProperty]] = V.[[TargetProperty]]`
   c. Let `n = V.[[Length]]`
   d. If `n == 0`, Return
   e. For `j = 0` to `V.[[Length-1]]`, let `x[i + j] = V[j]`
4. Else [Note: `Type(V)` is XML]
   a. Let `x.[[TargetObject]] = V.[[Parent]]`
   b. If `V.[[Class]] == "processing-instruction"`, let `x.[[TargetProperty]] = null`
   c. Else let `x.[[TargetProperty]] = V.[[Name]]`
   d. Let the value of property `i` of `x` be `V`
5. Let `x.[[Length]] = x.[[Length]] + n`
6. Return

8.2.1.7 `[[DeepCopy]]` ( )

**Overview**

The XMLList type adds the internal `[[DeepCopy]]` method to the internal properties defined by the Object type. The XMLList `[[DeepCopy]]` method is used to create and return a copy of this XMLList object containing deep copies of all its properties.

**Semantics**

When the `[[DeepCopy]]` method of an XMLList object `x` is called the following steps are taken:

1. Let `l` be a new XMLList object
2. Copy all internal properties of `x` to `l`
3. For `i = 0` to `x.[[Length-1]]`
   a. Let `l[i]` be the result of calling the `[[DeepCopy]]` method of `x[i]`
4. Return `l`

8.2.1.8 `[[Descendants]]` (P)

**Overview**

The XMLList type adds the internal `[[Descendants]]` method to the internal properties defined by the Object type. The XMLList `[[Descendants]]` method may be used to retrieve all the XML valued descendants of the properties in this XMLList (i.e., children, grandchildren, great-grandchildren, etc.) with names matching the input variable `P`. The input variable `P` may be a numeric property name, an unqualified name for an XML attribute (distinguished from the name of XML elements by a leading “@” symbol) or a set of XML elements, a QName for a set of XML elements, an AttributeName for a set of XML attributes, the properties wildcard “*” or the attributes wildcard “@*”. When the input variable `P` is an unqualified XML element name, it identifies XML elements in the default namespace. When the input variable `P` is an unqualified XML attribute name, it identifies XML attributes in no namespace.

**Semantics**

When the `[[Descendants]]` method of an XML object `x` is called with property name `P`, the following steps are taken:

1. Let `l` be a new XMLList with `l.[[TargetObject]] = null`
2. For each property `q` in `x`
   a. If `x[q].[[Class]] == "element"`)
      i. Let `dq` be the result of calling the `[[Descendants]]` method of `x[q]` with argument `P`
      ii. If `dq.[[Length]] > 0`, call the `[[Append]]` method of `l` with argument `dq`
3. Return `l`
8.2.1.9 \([\text{Equals}]\) (V)

Overview

The XMLList type adds the internal \([\text{Equals}]\) method to the internal properties defined by the Object type. The XMLList \([\text{Equals}]\) method is used to compare this XMLList object for content equality with another XMLList object \(V\) or determine whether this XMLList object contains an object that compares equal to \(V\). The \([\text{Equals}]\) operator returns \text{true} if this XMLList object is considered equal to \(V\) or contains an object considered equal to \(V\). Otherwise, it returns \text{false}. The input variable \(V\) may be a value of type XMLList, XML or any value that can be converted to a String with \text{ToString()}.

Semantics

When the \([\text{Equals}]\) method of an XML object \(x\) is called with value \(V\), the following steps are taken:

1. If \(V\) == \text{undefined} and \(x.\text{[Length]}\) == 0, return \text{true}
2. If Type(V) is XMLList
   a. If \(V.\text{[Length]}\) == 1, call the \([\text{Equals}]\) method of \(x\) with argument \(V[0]\) and return the result
   b. If \(x.\text{[Length]}\) == 1, call the \([\text{Equals}]\) method of \(V\) with argument \(x[0]\) and return the result
   c. If \(x.\text{[Length]}\) is not equal to \(V.\text{[Length]}\), return \text{false}
   d. For \(i = 0\) to \(x.\text{[Length]}\)
      i. If the result of the comparison \(x[i] \equiv V[i]\) is \text{false}, return \text{false}
   e. Return \text{true}
3. For \(i = 0\) to \(x.\text{[Length]}\)-1
   a. If the result of the comparison \(x[i] \equiv V\) is \text{true}, return \text{true}
4. Return \text{false}

8.2.1.10 \([\text{ResolveValue}]\) ( )

Overview

The XMLList type adds the internal \([\text{ResolveValue}]\) method to the internal properties defined by the Object type. The XMLList \([\text{ResolveValue}]\) method is used to resolve the value of empty XMLLists. If this XMLList object is not empty, the \([\text{ResolveValue}]\) method will return it. If this XMLList is empty, the \([\text{ResolveValue}]\) method will attempt to create it based on the \([\text{TargetObject}]\) and \([\text{TargetProperty}]\) properties. If the XMLList cannot be created, \([\text{ResolveValue}]\) returns \text{null}.

Semantics

When the \([\text{ResolveValue}]\) method of an XMLList object \(x\) is called, the following steps are taken:

1. If \(x.\text{[Length]}\) > 0, return \(x\)
2. Else
   a. If \(x.\text{[TargetObject]} == \text{null}\) or (type(x.\text{[TargetProperty]})) is AttributeName)
      or (x.\text{[TargetProperty]} == \text{null}) or (x.\text{[TargetProperty]}\text{.localName} == \\
      "*")
      i. Return \text{null}
   b. Let \(base\) be the result of calling the \([\text{ResolveValue}]\) method of \(x.\text{[TargetObject]}\) recursively
   c. If \(base == \text{null}\), return \text{null}
   d. Let \(target\) be the result of calling \([\text{Get}]\) on \(base\) with argument \(x.\text{[TargetProperty]}\)
   e. If \(target.\text{[Length]}\) == 0
      i. If Type(base) is XMLList and (base.\text{[Length]} > 1), return \text{null}
      ii. Call \([\text{Put}]\) on \(base\) with arguments \(x.\text{[TargetProperty]}\) and the empty string
      iii. Let \(target\) be the result of calling \([\text{Get}]\) on \(base\) with argument \(x.\text{[TargetProperty]}\)
   f. Return \(target\)

8.3 The AttributeName Type

The internal AttributeName type is not a language data type. It is defined by this specification purely for expository purposes. An implementation of E4X must behave as if it produced and operated upon AttributeNames in the manner described here. However, a value of type AttributeName is used only as an intermediate result of expression evaluation and cannot be stored as the value of a variable or property.
The AttributeName type specifies the name of an XML attribute. A value of type AttributeName may be specified using an AttributeIdentifier. If the name of the attribute is not specified as a QualifiedIdentifier, the uri property of the associated QName will be the empty string representing no namespace.

8.3.1 Internal Properties
The following table summarises the internal properties of the AttributeName type.

<table>
<thead>
<tr>
<th>Property</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Name]]</td>
<td>None</td>
<td>The name of the attribute</td>
</tr>
</tbody>
</table>

The value of the [[Name]] property is a value of type QName.

8.4 The AnyName Type

The internal AnyName type is not a language data type. It is defined by this specification purely for expository purposes. An implementation of E4X must behave as if it produced and operated upon AnyNames in the manner described here. However, a value of type AnyName is used only as an intermediate result of expression evaluation and cannot be stored as the value of a variable or property.

The AnyName type is a marker type used to indicate that the name of a particular property was specified as a WildcardIdentifier (i.e., “*”). The AnyName type has exactly one value, called anyname.
9 Type Conversion

E4X extends the automatic type conversion operators defined in ECMAScript. Note: as in ECMAScript Edition 3, these type conversion functions are internal and are not directly accessible by users. They occur as needed in E4X algorithms and are described here to aid specification of type conversion semantics. In addition, ToString and ToXMLString are exposed indirectly to the E4X user via the built-in methods toString() and toXMLString() defined in sections 12.4.4.36, 12.4.4.37, 12.5.4.18 and 12.5.4.19.

9.1 ToString

E4X extends the behaviour of the ToString operator by specifying its behaviour for the following types.

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML</td>
<td>Return the XML object as a string as defined in section 9.1.1.</td>
</tr>
<tr>
<td>XMLList</td>
<td>Return the XMLList object as a string as defined in section 9.1.2.</td>
</tr>
</tbody>
</table>

9.1.1 ToString Applied to the XML Type

Overview

Given an XML object x, the operator ToString converts x to a string s. If a value of type XML has simple content (i.e., contains no elements), it represents a primitive value and ToString returns the String contents of the XML object, omitting the start tag, attributes, namespace declarations and end tag. Otherwise, ToString returns an XML encoded string representing the entire XML object, including the start tag, attributes, namespace declarations and the end tag.

Combined with ToString's treatment of XMLLists (see section 9.1.2), this behaviour allows E4X programmers to access the values of XML leaf nodes in much the same way they access the values of object properties. For example, given a variable named order assigned to the following XML value:

```xml
<order>
  <customer>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
  </customer>
  <item>
    <description>Big Screen Television</description>
    <price>1299.99</price>
    <quantity>1</quantity>
  </item>
</order>
```

the E4X programmer can access individual values of the XML value like this:

```javascript
// Construct the full customer name
var name = order.customer.firstname + " " + order.customer.lastname;

// Calculate the total price
var total = order.item.price * order.item.quantity;
```

E4X does not require the programmer to explicitly select the text nodes associated with each leaf element or explicitly select the first element of each XMLList return value. For cases where this is not the desired behaviour, the ToXMLString operator is provided (see section 9.2). Note: in the example above, the String valued properties associated with the XML objects order.item.price and order.item.quantity are implicitly converted to type Number prior to performing the multiply operation.

For XML objects with [[Class]] set to “attribute” or “text”, ToString simply returns their value as a string.
Semantics

Given an XML object $x$, ToString takes the following steps:

1. If $x.[[\text{Class}]] \in \{"attribute", "text"\}$, return $x.[[\text{Value}]]$
2. If $x$.hasSimpleContent() == true
   a. Let $s$ be the empty string
   b. For $i = 0$ to $x.[[\text{Length}]]-1$,
      i. If $x[i].[[\text{Class}]] \notin \{"comment", "processing-instruction"\}$
         1. Let $s$ be the result of concatenating $s$ and ToString($x[i]$)
   c. Return $s$
3. Else
   a. Return ToXMLString($x$)

9.1.2 ToString Applied to the XMLList Type

Overview

The operator ToString converts an XMLList object $l$ to a string $s$. The return value is the string representation of each item in the XMLList concatenated together in order.

Note that the result of calling ToString on a list of size one is identical to the result of calling ToString on the single item contained in the XMLList. This treatment intentionally blurs the distinction between a single XML object and an XMLList containing only one value to simplify the programmer’s task. It allows E4X programmers to access the value of an XMLList containing only a single primitive value in much the same way they access object properties.

Semantics

Given an XMLList object $l$, ToString performs the following steps:

1. If $l$.hasSimpleContent() == true
   a. Let $s$ be the empty string
   b. For $i = 0$ to $l.[[\text{Length}]]-1$,
      i. If $x[i].[[\text{Class}]] \notin \{"comment", "processing-instruction"\}$
         1. Let $s$ be the result of concatenating $s$ and ToString($l[i]$)
   c. Return $s$
2. Else
   a. Return ToXMLString($x$)

9.2 ToXMLString ( input argument, [AncestorNamespaces], [IndentLevel] )

E4X adds the conversion operator ToXMLString to ECMAScript. ToXMLString is a variant of ToString used to convert its argument to an XML encoded string. Unlike ToString, it always includes the start tag, attributes, namespace declarations and end tag associated with an XML element, regardless of content. This is useful in cases where the default ToString behaviour is not desired. The semantics of ToXMLString are specified by the following table.

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a TypeError exception.</td>
<td></td>
</tr>
<tr>
<td>Null</td>
<td>Throw a TypeError exception.</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>Return ToString(input argument)</td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>Return ToString(input argument)</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>Return EscapeElementValue(input argument)</td>
<td></td>
</tr>
<tr>
<td>XML</td>
<td>Create an XML encoded string value based on the content of the XML object as specified in section 9.2.1.</td>
<td></td>
</tr>
<tr>
<td>XMLList</td>
<td>Create an XML encoded string value by calling ToXMLString on each property of the XMLList in order passing the optional argument AncestorNamespaces and concatenating the results to form a single string.</td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>Apply the following steps:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. Let $p$ be the result of calling ToPrimitive(input argument, hint String)</td>
<td></td>
</tr>
</tbody>
</table>
9.2.1 ToXMLString Applied to the XML Type

Semantics

Given an XML object \( x \) and an optional argument \( \text{AncestorNamespaces} \) and an optional argument \( \text{IndentLevel} \), ToXMLString converts it to an XML encoded string \( s \) by taking the following steps:

1. Let \( s \) be the empty string
2. If \( \text{IndentLevel} \) was not provided, Let \( \text{IndentLevel} = 0 \)
3. If \( \text{XML.prettyPrinting} == \text{true} \)
   a. For \( i = 0 \) to \( \text{IndentLevel}-1 \), let \( s \) be the result of concatenating \( s \) and the space \(<\text{SP}>\) character
4. If \( x.\{\text{Class}\} == \"text\" \),
   a. If \( \text{XML.prettyPrinting} == \text{true} \)
      i. Let \( \nu \) be the result of removing all the leading and trailing whitespace characters from \( x.\{\text{Value}\} \)
      ii. Return \( \text{EscapeElementValue}(\nu) \)
   b. Else
      i. Return \( \text{EscapeElementValue}(x.\{\text{Value}\}) \)
5. If \( x.\{\text{Class}\} == \"attribute\" \), return \( \text{EscapeAttributeValue}(x.\{\text{Value}\}) \)
6. If \( x.\{\text{Class}\} == \"comment\" \), return the result of concatenating the string \"<!\!--, \( x.\{\text{Value}\} \) and the string \"--\>
7. If \( x.\{\text{Class}\} == \"processing-instruction\" \), return the result of concatenating the string \"<?", \( x.\{\text{Name}\}.\text{localName} \), the space \(<\text{SP}>\) character, \( x.\{\text{Value}\} \) and the string \"?>\"
8. If \( \text{AncestorNamespaces} \) was not provided, let \( \text{AncestorNamespaces} = \{\} \)
9. Let \( \text{namespaceDeclarations} = \{\} \)
10. For each \( ns \) in \( x.\{\text{InScopeNamespaces}\} \)
    a. If there is no \( \text{ns} \in \{\text{AncestorNamespaces}\} \), such that \( \text{ns}.\text{uri} == \text{ns}.\text{uri} \) and \( \text{ns}.\text{prefix} == \text{ns}.\text{prefix} \)
       i. Let \( \text{namespaceDeclarations} = \text{namespaceDeclarations} \cup \text{ns} \)
       Note: implementations may also exclude unused namespace declarations from \( \text{namespaceDeclarations} \)
11. Let \( \text{namespace} \) be the result of calling \( [[\text{GetNamespace}]] \) on \( x.\{\text{Name}\} \) with argument \( \{\text{AncestorNamespaces} \cup \text{namespaceDeclarations}\} \)
12. If \( \text{namespace.prefix} == \text{undefined} \),
    a. Let \( \text{namespace.prefix} \) be an arbitrary implementation defined namespace prefix, such that there is no \( \text{ns} \in \{\text{AncestorNamespaces} \cup \text{namespaceDeclarations}\} \) with \( \text{ns}.\text{prefix} == \text{ns}.\text{prefix} \)
    b. If there is no \( \text{ns} \in \{\text{AncestorNamespaces} \cup \text{namespaceDeclarations}\} \), such that \( \text{ns}.\text{uri} == \text{namespace.uri} \) and \( \text{ns}.\text{prefix} == \text{namespace.prefix} \),
       i. Call \( [[\text{AddInScopeNamespace}]\text{ns}] \) on \( x \) with argument \( \text{namespace} \)
       ii. Let \( \text{namespaceDeclarations} = \text{namespaceDeclarations} \cup \text{namespace} \)
13. Let \( s \) be the result of concatenating \( s \) and the string \"<\"
14. If \( \text{namespace.prefix} \) is not the empty string,
    a. Let \( s \) be the result of concatenating \( s, \text{namespace.prefix} \) and the string \".:\"
15. Let \( s \) be the result of concatenating \( s \) and \( x.\{\text{Name}\}.\text{localName} \)
16. Let \( \text{attrAndNamespaces} = x.\{\text{Attributes}\} \cup \text{namespaceDeclarations} \)
17. For each \( \text{an} \) in \( \text{attrAndNamespaces} \)
    a. Let \( s \) be the result of concatenating \( s \) and the space \(<\text{SP}>\) character
    b. If \( \text{Type}(\text{an}) \) is XML and \( \text{an.}\{\text{Class}\} == \"attribute\"
       i. Let \( \text{ans} \) be the result of calling \( [[\text{GetNamespace}]] \) on \( a.\{\text{Name}\} \) with argument \( \text{AncestorNamespaces} \)
       ii. If \( \text{ans}.\text{prefix} == \text{undefined} \),
          1. Let \( \text{ans}.\text{prefix} \) be an arbitrary implementation defined namespace prefix, such that there is no \( \text{ns} \in \{\text{AncestorNamespaces} \cup \text{namespaceDeclarations}\} \) with \( \text{ans}.\text{prefix} == \text{ns}.\text{prefix} \)
          2. If there is no \( \text{ns} \in \{\text{AncestorNamespaces} \cup \text{namespaceDeclarations}\} \), such that \( \text{ns}.\text{uri} == \text{ans}.\text{uri} \) and \( \text{ns}.\text{prefix} == \text{ans}.\text{prefix} \)
             a. Call \( [[\text{AddInScopeNamespace}]]\text{ns} \) on \( x \) with argument \( \text{ans} \)
             b. Let \( \text{namespaceDeclarations} = \text{namespaceDeclarations} \cup \text{ans} \)
          iii. If \( \text{ans}.\text{prefix} \) is not the empty string
             1. Let \( s \) be the result of concatenating \( s, \text{namespace.prefix} \) and the string \".:\"
        iv. Let \( s \) be the result of concatenating \( s \) and \( a.\{\text{Name}\}.\text{localName} \)
c. Else
   i. Let s be the result fo concatenating s and the string "xmlns"
   ii. If (an.prefix == undefined),
       1. Let an.prefix be an arbitrary implementation defined namespace prefix, such that there is no ns2 ∈ (AncestorNamespaces ∪ namespaceDeclarations) with an.prefix == ns2.prefix
   iii. If an.prefix is not the empty string
       1. Let s be the result of concatenating s, the string ",:" and an.prefix
   d. Let s be the result of concatenating s, the string ",=" and a double-quote character (i.e. Unicode codepoint \u0022)
   e. If an.[[Class]] == "attribute"
      i. Let s be the result fo concatenating s and an.[[Value]]
   f. Else
      i. Let s be the result of concatenating s and an.uri
      g. Let s be the result of concatenating s and a double-quote character (i.e. Unicode codepoint \u0022)
18. If x.[[Length]] == 0
   a. Let s be the result of concatenating s and "/>
   b. If (XML.prettyPrinting == true), let s be the result of concatenating s and a LineTerminator
   c. Return s
19. Let s be the result of concatenating s and the string ">"?
20. Let indentChildren = (x.[[Length]] > 1) or (x.[[Length]] == 1 and x[0].[[Class]] is not equal to "text")
21. If (XML.prettyPrinting == true and indentChildren == true)
   a. Let s be the result of concatenating s and LineTerminator
   b. Let nextIndentLevel = IndentLevel + XML.PrettyIndent.
22. Else
   a. Let nextIndentLevel = 0
23. For i = 0 to x.[[Length]]-1
   a. Let child = ToXMLString (x[i], (AncestorNamespaces ∪ namespaceDeclarations), nextIndentLevel)
   b. Let s be the result of concatenating s and child
24. If (XML.prettyPrinting == true and indentChildren == true),
   a. Let s be the result of concatenating s and a LineTerminator
   b. For i = 0 to IndentLevel, let s be the result of concatenating s and a space <SP> character
25. Let s be the result of concatenating s and the string "</n
26. If namespace.prefix is not the empty string
   a. Let s be the result of concatenating s, namespace.prefix and the string ":[
27. Let s be the result of concatenating s, x.[[Name]].localName and the string ">"
28. If (XML.prettyPrinting == true), let s be the result of concatenating s and a LineTerminator
29. Return s

Note: implementations may also preserve insignificant whitespace (e.g., inside and between element tags) and attribute quoting conventions in ToXMLString().

Given a String value s, the operator EscapeElementValue performs the following steps:

1. Let r be the empty string
2. For each character c in s
   a. If (c == "\n"), let r be the result of concatenating r and the string "&lt;"
   b. Else if (c == "\>"), let r be the result of concatenating r and the string "&gt;"
   c. Else if (c == "\"), let r be the result of concatenating r and the string "&amp;"
   d. Else, let r be the result of concatenating r and c
3. Return r

Given a string value s, the operator EscapeAttributeValue performs the following steps:

1. Let r be the empty string
2. For each character c in s
   a. If (c is a double quote character (i.e., ")), let r be the result of concatenating r and the string "&quot;"
   b. Else if (c == "\n") let r be the result of concatenating r and the string "&lt;"
   c. Else if (c == "\") let r be the result of concatenating r and the string "&amp;"
   d. Else if (c == \u000A) let r be the result of concatenating r and the string "&#xA;"
e. Else if (c == \u000D) let r be the result of concatenating r and the string "\n"

f. Else if (c == \u0009) let r be the result of concatenating r and the string "\t"

g. Else let r be the result of concatenating r and c

3. Return r

9.3 ToXML

E4X adds the operator ToXML to ECMAScript. ToXML converts its argument to a value of type XML according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Convert the input argument to a string using ToString then convert the result to XML as specified in section 9.3.1.</td>
</tr>
<tr>
<td>Number</td>
<td>Convert the input argument to a string using ToString then convert the result to XML as specified in section 9.3.1.</td>
</tr>
<tr>
<td>String</td>
<td>Create an XML object from the String as specified below in section 9.3.1.</td>
</tr>
<tr>
<td>XML</td>
<td>Return the input argument (no conversion).</td>
</tr>
<tr>
<td>XMLList</td>
<td>If the XMLList contains only one property and the type of that property is XML, return that property. Otherwise, throw a TypeError exception.</td>
</tr>
<tr>
<td>Object</td>
<td>If the [Class] property of the input argument is &quot;String&quot;, &quot;Number&quot; or &quot;Boolean&quot;, convert the input argument to a string using ToString then convert the result to XML as specified in section 9.3.1. Otherwise, throw a TypeError exception.</td>
</tr>
<tr>
<td>W3C XML Information Item</td>
<td>Create an XML object from a W3C XML Information Item as specified below in section 9.3.2.</td>
</tr>
</tbody>
</table>

9.3.1 ToXML Applied to the String Type

**Overview**

When ToXML is applied to a string type, it converts it to XML by parsing the string as XML. Prior to conversion, string concatenation can be used to construct portions of the XML object without regard for XML constraints such as well-formedness. For example, consider the following.

```javascript
var John = "<employee><name>John</name><age>25</age></employee>";
var Sue = "<employee><name>Sue</name><age>32</age></employee>";
var tagName = "employees";
var employees = new XML("\" + tagName + \\\"\" + John + Sue + "\" + tagName + \\\"\"");
```

**Semantics**

Given a String value s, ToXML converts the string to an XML object using the following steps:

1. Let defaultNamespace = GetDefaultNamespace()
2. Let parentString be the result of concatenating the strings "<parent xmlns=", defaultNamespace, ", ">", s and ">
3. Parse parentString as a W3C Element Information Item e and if the parse fails, throw a SyntaxError exception
4. Let x = ToXML(e)
5. If x.[[Length]] == 0
   a. Return a new XML object t with t.[[Class]] = "text", t.[[Parent]] = null and t.[[Value]] = the empty string
6. else if x.[[Length]] == 1
   a. Let x[0].[[Parent]] == null
   b. Return x[0]
7. else throw a SyntaxError exception

Note: the use of a W3C XML Information Item is purely illustrative. A W3C XML Information Item is not required to perform this type conversion and implementations may use any mechanism that provides the same semantics.
9.3.2 ToXML Applied to a W3C XML Information Item

Overview

When ToXML is applied to an implementation of a XML Information Item conforming to the XML Information Set specification, it maps the E4X data model onto the given information item such that E4X operators may be used to query, navigate and manipulate the given information item.

Implementations may expose this functionality directly to a user via the XML constructor; however, this is not required for conformance with E4X. For example, E4X might be deployed to facilitate manipulating the document object in a web browser as follows:

```javascript
function createTable() {
    var doc = XML(document); // create an E4X wrapper for the document
    var mytablebody = doc..body.TABLE.TBODY;

    for(j=0;j<2;j++) {
        mytablebody.TR[j] = ""; // append an empty table row
        for(i=0;i<2;i++) // append a cell with some content
            mytablebody.TR[j].TD[i] = "cell is row " + j + ", column " + i;
    }
    doc..body.TABLE.@border = 2; // set the border attribute of the table
}
```

Instead of writing the equivalent DOM code below.

```javascript
function createTable () {
    var mybody=document.getElementsByTagName("body").item(0);
    mytable = document.createElement("TABLE");
    mytablebody = document.createElement("TBODY");
    for(j=0;j<2;j++) {
        mycurrent_row=document.createElement("TR");
        for(i=0;i<2;i++) {
            mycurrent_cell=document.createElement("TD");
            currenttext=document.createTextNode("cell is row "+j+", column "+i);
            mycurrent_cell.appendChild(currenttext);
            mycurrent_row.appendChild(mycurrent_cell);
        }
        mytablebody.appendChild(mycurrent_row);
    }
    mytable.appendChild(mytablebody);
    mybody.appendChild(mytable);
    mytable.setAttribute("border","2");
}
```

Semantics

A W3C Information Item $i$ is mapped onto an XML type $x$ as follows:

1. Map $x.\{[\text{Parent}]\}$ to **null**
2. If $i$ is a character information item
   a. Map $x.\{[\text{Class}]\}$ to "text"
b. Map x.[[Value]] to the largest contiguous sequence of character information items that have the same parent starting with i and continuing forward in document order. Map each character of the string x.[[Value]] to the corresponding [character code] property of each character information item in the sequence.

c. If XML.ignoreWhitespace == true and (each character in x.[[Value]] is a Whitespace), return null

d. Else return x

3. if i is a comment information item
a. If XML.ignoreComments == true, return null
b. Map x.[[Class]] to "comment"
c. Map x.[[Value]] to the [content] property of i
d. Return x

4. If i is a processing instruction information item
a. If XML.ignoreProcessingInstructions == true, return null
b. Map x.[[Class]] to "processing-instruction"
c. Map x.[[Name]] to the [target] property of i
d. Map x.[[Value]] to the [content] property of i
e. Return x

5. If i is an attribute information item
a. Map x.[[Class]] to "attribute"
b. Map x.[[Name]].localName to the [local name] property of i
c. Map x.[[Name]].uri to the [namespace name] property of i
   Note: implementations may also map x.[[Name]].[[Prefix]] to the [prefix] property of i.
d. Map x.[[Value]] to the [normalized value] property of i
e. Return x

6. If i is an element information item
a. Map x.[[Class]] to "element"
b. Map x.[[Name]].localName to the [local name] property of i
c. Map x.[[Name]].uri to the [namespace name] property of i
   Note: implementations may also map x.[[Name]].[[Prefix]] to the [prefix] property of i.
d. For each attribute information item a in the [attributes] property of i
   i. Map a member attr of x.[[Attributes]] to the result of calling ToXML(a)
   ii. Map attr.[[Parent]] to x
   e. Note: The E4X data model does not enforce the constraint: \( \forall x \in XML : x.[[InScopeNamespaces]] \supseteq x.[[Parent]].[[InScopeNamespaces]] \). However, implementations may at this point map members of x.[[InScopeNamespaces]] to members of x.[[Parent]].[[InScopeNamespaces]].
   f. For each attribute information item a in the [namespace attributes] property of i
      i. Map a member ns of x.[[InScopeNamespaces]] to a as follows:
         1. if the [local name] property of a is "xmlns"
            a. Map ns.prefix to the empty string
         2. else
            a. Map ns.prefix to the [local name] property of a
      3. Map ns.uri to the [normalized value] property of a

g. Let J = 0
h. Let xmlChild = 0
i. Let numItemChildren be the number of information items in the [children] property of i
j. While (j < numItemChildren)
   i. Let item be the jth information item in the [children] property of i
   ii. Let c = ToXML(item)
   iii. If c is not null
      1. Map x[xmlChild] to c
      2. Map x[xmlChild].[[Parent]] to x
      3. if c.[[Class]] == "text"
         a. Let j = j + c.[[Value]].length - 1
      4. Let xmlChild = xmlChild + 1
   iv. Let j = j + 1
   k. Map x.[[Length]] to xmlChild
   l. Return x

7. if i is a document information item
   a. Return the result of calling ToXML on the [document element] property of i

8. if i is an unexpanded entity reference information item
a. Throw a `ReferenceError` exception

9. Return `null`
   Note: ToXML ignores document type declaration information items

### 9.4 `ToXMLList`

E4X adds the operator `ToXMLList` to ECMAScript. `ToXMLList` converts its argument to a value of type `XMLList` according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a <code>TypeError</code> exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a <code>TypeError</code> exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Convert the input argument to a string using <code>ToString</code> then convert the result to <code>XMLList</code> as specified in section 9.4.1.</td>
</tr>
<tr>
<td>Number</td>
<td>Convert the input argument to a string using <code>ToString</code> then convert the result to <code>XMLList</code> as specified in section 9.4.1.</td>
</tr>
<tr>
<td>String</td>
<td>Create an <code>XMLList</code> object from the String as specified below in section 9.4.1.</td>
</tr>
<tr>
<td>XML</td>
<td>Create an <code>XMLList</code> object <code>l</code> with <code>l.[[Length]] = 1</code>, <code>l.[[TargetObject]] = null</code> and <code>l[0] =</code> the input argument.</td>
</tr>
<tr>
<td>XMLList</td>
<td>Return the input argument (no conversion).</td>
</tr>
<tr>
<td>Object</td>
<td>If the <code>[[Class]]</code> property of the input argument is &quot;String&quot;, &quot;Number&quot; or &quot;Boolean&quot;, convert the input argument to a string using <code>ToString</code> then convert the result to <code>XML</code> as specified in section 9.3.1. Otherwise, throw a <code>TypeError</code> exception.</td>
</tr>
</tbody>
</table>

#### 9.4.1 `ToXMLList` Applied to the String Type

**Overview**

When `ToXMLList` is applied to a string type, it converts the string type to an `XMLList` by parsing the string as an XML fragment. Prior to conversion, string concatenation can be used to construct portions of the `XMLList` value. For example,

```javascript
var John = "<employee><name>John</name><age>25</age></employee>";
var Sue = "<employee><name>Sue</name><age>32</age></employee>";
var l = new XMLList(John + Sue);
```

**Semantics**

Given a String value `s`, `ToXMLList` converts it to an `XMLList` using the following steps:

1. Let `defaultNamespace = GetDefaultNamespace()`.
2. Let `parentString` be the result of concatenating the strings "<parent xmlns='', defaultNamespace, "">", `s` and "</parent>";
3. Parse `parentString` as a W3C Element Information Item `e`.
4. If the parse fails, throw a `SyntaxError` exception.
5. Let `x = ToXML(e)`.
6. Let `l` be a new `XMLList` with `l.[[TargetObject]] = null`.
7. For `i = 0` to `x.[[Length]]-1`
   a. Let `x[i].[Parent] = null`
   b. Call the `[[Append]]` method of `l` with argument `x[i]`.
8. Return `l`.

Note: the use of a W3C XML Information Item is purely illustrative. A W3C XML Information Item is not required to perform this type conversion and implementations may use any mechanism that provides the same semantics.
9.5 ToAttributeName

E4X adds the operator ToAttributeName to ECMAScript. ToAttributeName converts its argument to a value of type AttributeName according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Number</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>String</td>
<td>Create an AttributeName from the String as specified below in section 9.5.1</td>
</tr>
<tr>
<td>XML</td>
<td>Convert the input argument to a string using ToString then convert the result to an AttributeName as specified in section 9.5.1.</td>
</tr>
<tr>
<td>XMLList</td>
<td>Convert the input argument to a string using ToString then convert the result to an AttributeName as specified in section 9.5.1.</td>
</tr>
<tr>
<td>Object</td>
<td>If the input argument is a QName object (i.e., its internal [[Class]] property is &quot;QName&quot;), return a new AttributeName with its [[Name]] property set to the input argument. Otherwise, convert the input argument to a string using ToString then convert the result to an AttributeName as specified in section 9.5.1.</td>
</tr>
<tr>
<td>AttributeName</td>
<td>Return the input argument (no conversion).</td>
</tr>
<tr>
<td>AnyName</td>
<td>Return the result of calling ToAttributeName(&quot;**&quot;)</td>
</tr>
</tbody>
</table>

9.5.1 ToAttributeName Applied to the String Type

Given a string s, the ToAttributeName conversion function returns an AttributeName a. The [[Name]] property of a is set to a new QName q with its local name set to the given string and its URI set to the empty string representing no namespace.

**Semantics**

Given a String value s, ToAttributeName converts it to an AttributeName using the following steps:

1. Let ns be a new Namespace created as if by calling the constructor new Namespace()
2. Let q be a new QName created as if by calling the constructor new QName(ns, s)
3. Return a new AttributeName a with a.[[Name]] = q

9.6 ToXMLName

E4X adds the operator ToXMLName to ECMAScript. ToXMLName is an internal operator that converts its argument to a value of type AttributeName or a QName object according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Number</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>String</td>
<td>Create a QName object or AttributeName from the String as specified below in section 9.6.1.</td>
</tr>
<tr>
<td>XML</td>
<td>Convert the input argument to a string using ToString then convert the result to a QName object or AttributeName as specified in section 9.6.1.</td>
</tr>
<tr>
<td>XMLList</td>
<td>Convert the input argument to a string using ToString then convert the result to a QName object or AttributeName as specified in section 9.6.1.</td>
</tr>
<tr>
<td>Object</td>
<td>If the input argument is a QName object (i.e., its [[Class]] property is &quot;QName&quot;), return the input argument. Otherwise, convert the input argument to a string using ToString then convert the result to a QName object or AttributeName as specified in section 9.6.1.</td>
</tr>
<tr>
<td>AttributeName</td>
<td>Return the input argument (no conversion).</td>
</tr>
<tr>
<td>AnyName</td>
<td>Return the result of calling ToXMLName(&quot;**&quot;)</td>
</tr>
</tbody>
</table>
9.6.1 ToXMLName Applied to the String Type

Given a string `s`, the ToXMLName conversion function returns a QName object or AttributeName. If the first character of `s` is "@", ToXMLName creates an AttributeName using the ToAttributeName operator. Otherwise, it creates a QName object using the QName constructor.

Semantics

Given a String value `s`, ToXMLName operator converts it to a QName object or AttributeName using the following steps:

1. If the first character of `s` is "@"
   a. Let `name = s.substring(1, s.length)`
   b. Return ToAttributeName(`name`)
2. Else
   a. Return a QName object created as if by calling the constructor new QName(`s`)
10 Expressions

10.1 Primary Expressions

Syntax

E4X extends the primary expressions defined by ECMAScript with the following production:

\[
\text{PrimaryExpression} : \text{PropertyIdentifier} \\
\text{XMLInitialiser} \\
\text{XMLListInitialiser}
\]

\[
\text{PropertyIdentifier} : \text{AttributeIdentifier} \\
\text{QualifiedIdentifier} \\
\text{WildcardIdentifier}
\]

Semantics

The production PrimaryExpression : PropertyIdentifier is evaluated as follows:

1. Let \( n \) be the result of evaluating PropertyIdentifier
2. Let \( name = \text{ToXMLName}(n) \)
3. While (true)
   a. If there are no more objects on the scope chain,
      i. Return undefined
   b. Let \( o \) be the next object on the scope chain.
   c. If Type(\( o \)) \( \in \{ \text{XML, XMLList} \} \)
      i. Let hasProp be the result of calling the \([\text{HasProperty}]\) method of \( o \), passing \( name \) as the property
      ii. If hasProp \( \equiv \text{true} \)
         1. Return a value of type Reference whose base object is \( o \) and whose property name is \( name \)

The production PropertyIdentifier : AttributeIdentifier is evaluated as follows:

1. Return the result of evaluating AttributeIdentifier

The productions PropertyIdentifier : QualifiedIdentifier, and PropertyIdentifier : WildcardIdentifier are evaluated exactly the same manner except AttributeIdentifier is replaced by QualifiedIdentifier and WildcardIdentifier in step 1.

10.1.1 Attribute Identifiers

Syntax

E4X extends ECMAScript by adding attribute identifiers. The syntax of an attribute identifier is specified by the following production:

\[
\text{AttributeIdentifier} : \@ \text{PropertySelector} \\
\@ \text{QualifiedIdentifier} \\
\@ [ \text{Expression} ]
\]

\[
\text{PropertySelector} : \text{Identifier} \\
\text{WildcardIdentifier}
\]

Overview
An `AttributeIdentifier` is used to identify the name of an XML attribute. It evaluates to a value of type `AttributeName`. The preceding “@” character distinguishes a XML attribute from a XML element with the same name. This `AttributeIdentifier` syntax was chosen for consistency with the familiar XPath syntax.

**Semantics**

The production `AttributeIdentifier : @ PropertySelector` is evaluated as follows:

1. Let `name` be a string value containing the same sequence of characters as in the `PropertySelector`
2. Return `ToAttributeName(name)`

The production `AttributeIdentifier : @ QualifiedIdentifier` is evaluated as follows:

1. Let `q` be the result of evaluating `QualifiedIdentifier`
2. Return `ToAttributeName(q)`

The production `AttributeIdentifier : @ [ Expression ]` is evaluated as follows:

1. Let `e` be the result of evaluating `Expression`
2. Return `ToAttributeName(GetValue(e))`

### 10.1.2 Qualified Identifiers

**Syntax**

E4X extends ECMAScript by adding qualified identifiers. The syntax for qualified identifiers was chosen for consistency with future versions of ECMAScript and is specified by the following productions:

```
QualifiedIdentifier :
    PropertySelector :: PropertySelector
    PropertySelector :: [ Expression ]
```

**Overview**

`QualifiedIdentifiers` are used to identify values defined within a specific namespace. They may be used to access, manipulate and create namespace qualified XML element and attribute names. For example,
// Create a SOAP message
var message = {
  soap:Envelope {
    xmlns:soap="http://schemas.xmlsoap.org/soap-envelope/",
    soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  },
  soap:Body {
    <m:GetLastTradePrice xmlns:m="http://mycompany.com/stocks">
      <symbol>DIS</symbol>
    </m:GetLastTradePrice>
  }
}

// declare the SOAP and stocks namespaces
var soap = new Namespace("http://schemas.xmlsoap.org/soap-envelope/");
var stock = new Namespace("http://mycompany.com/stocks");

// extract the soap encoding style and body from the soap message
var encodingStyle = message.soap::encodingStyle;
var body = message.soap::Body;

// change the stock symbol
message.soap::Body.stock::GetLastTradePrice.symbol = "MYCO";

Semantics

A QualifiedIdentifier evaluates to a QName object. The production QualifiedIdentifier : PropertySelector :: PropertySelector is evaluated as follows:

1. Let ns be the result of evaluating the first PropertySelector
2. if (ns == anyname), let ns = null
3. Let localName be a string value containing the same sequence of characters as in the second PropertySelector
4. Return a new QName created as if by calling the constructor new QName(GetValue(ns), localName)

The production QualifiedIdentifier : PropertySelector :: [ Expression ] is evaluated as follows:

1. Let ns be the result of evaluating PropertySelector
2. if (ns == anyname), let ns = null
3. Let e be the result of evaluating Expression
4. Return a new QName created as if by calling the constructor new QName(GetValue(ns), GetValue(e))

10.1.3 Wildcard Identifiers

Syntax

E4X extends ECMAScript by adding a wildcard identifier. The syntax of the wildcard identifier is specified by the following production:

    WildcardIdentifier : *

Overview

The WildcardIdentifier is used to identify any name. It may be used for matching namespaces, properties of XML objects or XML attributes. The wildcard identifier evaluates to the value anyname.

Semantics

The production WildcardIdentifier : * is evaluated as follows:
1. Return **anyname**

### 10.1.4 XML Initialiser

#### Overview

An XML initialiser is an expression describing the initialization of an XML object, written in a form of a literal. It may specify an XML element, an XML comment, an XML PI, or a CDATA section using ordinary XML syntax. For XML elements, it provides the name, XML attributes and XML properties of an XML object.

The syntactic grammar for XML initialisers processes input elements produced by the lexical grammar goal symbols `InputElementXMLTag` and `InputElementXMLContent`. These input elements are described in section 7.2.

Below are some examples of XML initialisers.

```xml
// an XML object representing a person with a name and age
var person = <person><name>John</name><age>25</age></person>;

// a variable containing an XML object representing two employees
var e = <employees>
    <employee id="1"><name>Joe</name><age>20</age></employee>
    <employee id="2"><name>Sue</name><age>30</age></employee>
</employees>;
```

Expressions may be used to compute parts of an XML initialiser. Expressions are delimited by curly braces and may appear inside tags or element content. Inside a tag, expressions may be used to compute a tag name, attribute name, or attribute value. Inside an element, expressions may be used to compute element content. For example,

```javascript
for (i = 0; i < 10; i++)
    e[i] = <employee id={i}>
       <name>{names[i].toUpperCase()}</name>
       <age>{ages[i]}</age>
    </employee>;
```

Each expression is evaluated and replaced by its value prior to parsing the literal XML value. For example the following expression,

```javascript
var tagname = "name";
var attributename = "id";
var attributevalue = 5;
var content = "Fred";

var x = <{tagname} {attributename}={attributevalue}>{content}</{tagname}>;
```

would assign the following XML value to the variable x.

```xml
<name id="5">Fred</name>
```

### Syntax

```
XMLInitialiser : XMLMarkup

XMLMarkup :
    XMLElement
```

XMLElement:
  < XMLTagContent />
  < XMLTagContent > XMLElementContent_opt </ XMLTagContent >

XMLTagContent:
  XMLTagCharacters XMLTagContent_opt
  { Expression } XMLTagContent_opt
  = WhiteSpace_opt { Expression } XMLTagContent_opt

XMLElementContent:
  XMLMarkup XMLElementContent_opt
  XMLText XMLElementContent_opt
  { Expression } XMLElementContent_opt

Semantics

The production XMLInitialiser : XMLMarkup is evaluated as follows:

1. Let markup be a string literal containing the same sequence of characters as XMLMarkup
2. Return a new XML object created as if by calling the XML constructor with argument markup (section 12.4.2)

The production XMLInitialiser : XMLElement is evaluated as follows:

3. Let element be a the result of evaluating XMLElement
4. Return a new XML object created as if by calling the XML constructor with argument element (section 12.4.2)

The production XMLElement : < XMLTagContent > is evaluated as follows:

1. Let content be the result of evaluating XMLTagContent
2. Return the result of concatenating the string value "<", followed by content, followed by the string value "/">

The production XMLElement : < XMLTagContent > XMLElementContent_opt </ XMLTagContent > is evaluated as follows:

1. Let startTag be the result of evaluating the first XMLTagContent
2. Let content be the result of evaluating XMLElementContent; if not present, use the empty string
3. Let endTag be the result of evaluating the second XMLTagContent
4. Return the result of concatenating the string value "<", followed by startTag, followed by the string value "/>", followed by content, followed by the string value ">", followed by endTag, followed by the string value "">

The production XMLTagContent : XMLTagCharacters XMLTagContent_opt is evaluated as follows:

1. Let tagChars be a string literal containing the same sequence of characters as XMLTagCharacters
2. Let tagContent be the result of evaluating XMLTagContent; if not present, use the empty string
3. Return the result of concatenating tagChars followed by tagContent

The production XMLTagContent : { Expression } XMLTagContent_opt is evaluated as follows:

1. Let expRef be the result of evaluating Expression
2. Let expression = GetValue(expRef)
3. Let tagContent be the result of evaluating XMLTagContent; if not present, use the empty string
4. Return the result of concatenating ToString(expression), followed by tagContent

The production XMLTagContent : = WhiteSpace_opt { Expression } XMLTagContent_opt is evaluated as follows:

1. Let expRef be the result of evaluating Expression
2. Let expression = GetValue(expRef)
3. Let attributeValue be the result of evaluating EscapeAttributeValue(ToString(expression))
4. Let tagContent be the result of evaluating XMLTagContent; if not present, use the empty string
5. Return the result of concatenating equals "=" followed by double-quote ", followed by \textit{attributeValue}, followed by double-quote ", followed by \textit{tagContent}

The production $\text{XMLElementContent} : \text{XMLMarkup} \text{XMLElementContent}_{opt}$ is evaluated as follows:

1. Let \textit{markup} be the result of evaluating \textit{XMLMarkup}
2. Let \textit{content} be the result of evaluating \textit{XMLElementContent}; if not present, use the empty string
3. Return the result of concatenating \textit{markup} followed by \textit{content}

The production $\text{XMLElementContent} : \text{XMLElement} \text{XMLElementContent}_{opt}$ is evaluated as follows:

1. Let \textit{element} be the result of evaluating \textit{XMLElement}
2. Let \textit{content} be the result of evaluating \textit{XMLElementContent}; if not present, use the empty string
3. Return the result of concatenating \textit{element} followed by \textit{content}

The production $\text{XMLElementContent} : \text{XMLText} \text{XMLElementContent}_{opt}$ is evaluated as follows:

1. Let \textit{text} be a string literal containing the same sequence of characters as \textit{XMLText}
2. Let \textit{content} be the result of evaluating \textit{XMLElementContent}; if not present, use the empty string
3. Return the result of concatenating \textit{text}, followed by \textit{content}

The production $\text{XMLElementContent} : \{ \text{Expression} \} \text{XMLElementContent}_{opt}$ is evaluated as follows:

1. Let \textit{expRef} be the result of evaluating \textit{Expression}
2. Let \textit{expression} = \textit{GetValue}\text{(expRef)}
3. If Type\text{(expression)} \in \{\text{XML, XMLList}\},
   a. Let \textit{value} be the result of calling ToXMLString\text{(expression)}
4. Else
   a. Let \textit{value} be the result of calling EscapeElementValue\text{(ToString\text{(expression)})}
5. Let \textit{content} be the result of evaluating \textit{XMLElementContent}; if not present, use the empty string
6. Return the result of concatenating \textit{value} followed by \textit{content}

10.1.5 XMLList Initialiser

Overview

An XMLList initialiser is an expression describing the initialization of an XMLList object written in a form resembling a literal. It describes an ordered list of XML properties using an anonymous XML element syntax. XMLList initialisers begin with the character sequence “<>” and end with the character sequence “</>”.

The syntactic grammar for XML initialisers processes input elements produced by the lexical grammar goal symbols \textit{InputElementXMLTag} and \textit{InputElementXMLContent}. These input elements are described in section 7.2.

Below are some examples of XMLList Initialisers,

\begin{verbatim}
var docfrag = <<<name>Phil</name><age>35</age><hobby>skiing</hobby>>/;
var emplist = <>
  <employee id="0" ><name>Jim</name><age>25</age></employee>
  <employee id="1" ><name>Joe</name><age>20</age></employee>
  <employee id="2" ><name>Sue</name><age>30</age></employee>
</>;
\end{verbatim}

Syntax

XMLListInitialiser : &lt; &gt; XMLElementContent &lt;/ &gt;
Semantics

The production XMLList : < > XMLElementContentopt </ > is evaluated as follows:

1. Let content be the result of evaluating XMLElementContent; if not specified use the empty string
2. Return a new XMLList object created as if by calling the XMLList constructor with argument content

10.2 Left-Hand-Side Expressions

E4X extends the left-hand-side expressions defined in ECMAScript with the following productions:

MemberExpression:
MemberExpression . PropertyIdentifier
MemberExpression .. Identifier
MemberExpression .. PropertyIdentifier
MemberExpression . ( Expression )

CallExpression:
CallExpression . PropertyIdentifier
CallExpression .. Identifier
CallExpression .. PropertyIdentifier
CallExpression . ( Expression )

In addition, E4X defines new semantics for existing left-hand-side expressions applied to values of type XML and XMLList.

10.2.1 Property Accessors

Syntax

E4X reuses and extends ECMAScript’s property accessor syntax for accessing properties and XML attributes within values of type XML and XMLList. XML properties may be accessed by name, using either the dot notation:

MemberExpression . Identifier
MemberExpression . PropertyIdentifier
CallExpression . Identifier
CallExpression . PropertyIdentifier

or the bracket notation:

MemberExpression [ Expression ]
CallExpression [ Expression ]

Overview

When MemberExpression or CallExpression evaluate to a XML object, the property accessor uses the XML [[Get]] method to determine the result. If the bracket notation is used with a numeric identifier, the XML [[Get]] method simply returns the property of the left operand with a property name matching the numeric identifier. Otherwise, the XML [[Get]] method examines the XML properties and XML attributes of the left operand and returns an XMLList containing the ones with names that match its right operand in order. For example,
```javascript
var order = <order id="123456" timestamp="Mon Mar 10 2003 16:03:25 GMT-0800 (PST)">
    <customer>
        <firstname>John</firstname>
        <lastname>Doe</lastname>
    </customer>
    <item>
        <description>Big Screen Television</description>
        <price>1299.99</price>
        <quantity>1</quantity>
    </item>
</order>;

var customer = order.customer; // get the customer element from the order
var id = order.@id; // get the id attribute from the order
var secondChild = order[1]; // get the second child element from the order by numeric index
var orderChildren = order.*; // get all the child elements from the order element
var orderAttributes = order.@*; // get all the attributes from the order element

When MemberExpression or CallExpression evaluate to an XMLList, the property accessor uses the XMLList [[Get]] method to determine the result. If the bracket notation is used with a numeric identifier, the XMLList [[Get]] method simply returns the property of the left operand with a property name matching the numeric identifier. Otherwise, the XMLList [[Get]] method applies the property accessor operation to each XML object in the list and returns a new XMLList containing the results in order. For example,

```javascript
var order = <order>
    <customer>
        <firstname>John</firstname>
        <lastname>Doe</lastname>
    </customer>
    <item id="3456">
        <description>Big Screen Television</description>
        <price>1299.99</price>
        <quantity>1</quantity>
    </item>
    <item id="56789">
        <description>DVD Player</description>
        <price>399.99</price>
        <quantity>1</quantity>
    </item>
</order>;

var descriptions = order.item.description; // get the list of all item descriptions
var itemIds = order.item.@id; // get the list of all item id attributes
var secondItem = order.item[1]; // get second item by numeric index
var itemChildren = order.item.*; // get the list of all child elements in all item elements
```

In the first property accessor statement above, the expression “order.item” examines the XML properties of the XML object bound to “order” and returns an XMLList containing the two named “item”. The expression “order.item.description” then examines the XML properties of each item in the resulting XMLList and returns an XMLList containing the two XML objects named “description”.

When MemberExpression or CallExpression do not evaluate to a value of type XML or XMLList and the right hand side of the expression is an Identifier, the property accessor performs according to the semantics specified in ECMAScript Edition 3. However, if MemberExpression or CallExpression do not evaluate to a value of type XML or XMLList and the right hand side of the expression is a PropertyIdentifier, the property accessor returns undefined.

Semantics

As in ECMAScript Edition 3, the behaviour of the production:

```
MemberExpression : MemberExpression . Identifier
```

is identical to the behaviour of the production:

```
MemberExpression : MemberExpression [ <identifier-string> ]
```

and similarly, the behaviour of the production:

```
CallExpression : CallExpression . Identifier
```

is identical to the behaviour of the production:

```
CallExpression : CallExpression [ <identifier-string> ]
```

where <identifier-string> is a string literal containing the same sequence of characters as the Identifier.

The production MemberExpression : MemberExpression [ Expression ] is evaluated as follows:

1. Let oRef be the result of evaluating MemberExpression
2. Let o = ToObject(GetValue(oRef))
3. Let pRef be the result of evaluating Expression
4. Let p = GetValue(pRef)
5. If (Type(o) ∈ {XML, XMLList})
   a. Return a value of type Reference whose base object is o and whose property name is ToXMLName(p)
6. Else
   a. If (Type(p) ∈ {AttributeName, AnyName}) or (Type(p) is Object and p.[[Class]] == "QName")
      i. Return undefined
   b. Return a value of type Reference whose base object is o and whose property name is ToString(p)

The production CallExpression : CallExpression [ Expression ] is evaluated in exactly the same manner, except the contained CallExpression is evaluated in step 1.

The production MemberExpression : MemberExpression . PropertyIdentifier behaves exactly as the production MemberExpression : MemberExpression [ Expression ], except the contained PropertyIdentifier is evaluated in step 3. Similarly, the production CallExpression : CallExpression . PropertyIdentifier behaves exactly as the production CallExpression : CallExpression [ Expression ], except the contained PropertyIdentifier is evaluated in step 3.

10.2.2 Function Calls

Syntax

E4X reuses ECMAScript's function call syntax for invoking methods on values of type XML and XMLList. The ECMAScript syntax for function calls is described by the following productions:

```
CallExpression :
  MemberExpression Arguments

Arguments :

  ()
  ( ArgumentList )
```
Overview

Unlike values of type Object, values of type XML and XMLList store and retrieve properties separately from methods so that XML method names do not clash with XML property names. For example,

```
var rectangle = <rectangle>
  <x>50</x>
  <y>75</y>
  <length>20</length>
  <width>30</width>
</rectangle>;
var numChildren = rectangle.length(); // returns 4 – number of children in <rectangle>
var rectangleLength = rectangle.length; // returns 20 – content of <length> element
rectangle.length = 50; // change the length element of the rectangle
```

To accomplish this, E4X modifies the semantics of the call expression to invoke the operation CallMethod (see section 10.2.2.1).

When the operation CallMethod is called with a single parameter r, it first checks to see if r is a Reference. If it is not, it attempts to call r as a function. However, if r is a Reference, it extracts the base and property name from the Reference r. Then, CallMethod calls the Object [[Get]] method to retrieve the property of the base object with the given property name. Note: The XML and XMLList [[Get]] method is never called for method lookup.

If no such property exists and base is an XMLList of size 1, CallMethod delegates the method invocation to the single property it contains. This treatment intentionally blurs the distinction between atomic objects and XMLLists of size 1.

If no such property exists and base is an XML object containing no XML valued children (i.e., an attribute, leaf node or element with simple content), CallMethod attempts to delegate the method lookup to the string value contained in the leaf node. This treatment allows users to perform operations directly on the value of a leaf node without having to explicitly select it. For example,

```
var shipto= <shipto>
  <name>Fred Jones</name>
  <street>123 Foobar Ave.</street>
  <citystatezip>Redmond, WA, 98008</citystatezip>
</shipto>
var upperName = shipto.name.toUpperCase(); // calls String.toUpperCase on value of text node
var citystatezip = shipto.citystatezip.split(", "); // calls String.split() on text node to parse address
var state = citystatezip[1]; // line into individual city, state and zip values
var zip = citystatezip[2];
```

Semantics

The production `CallExpression : MemberExpression Arguments` is evaluated as follows:

1. Let r be the result of evaluating `MemberExpression`
2. Let args be the result of evaluating Arguments, producing an internal list of argument values
3. Return the result of calling the operation CallMethod(r, args)

10.2.2.1 CallMethod (r, args)

Overview

The CallMethod abstract operator is used in this specification for invoking functions and methods. This operator is not part of the language. It is defined here to aid the specification of the language.

Semantics

Given a Reference r and a list of arguments args, the operation CallMethod performs the following steps:

1. Let f = r
2. Let base = null
3. If Type(r) is Reference
   a. Let base = GetBase(r)
   b. If base == null, throw a ReferenceException
   c. Let P = GetPropertyName(r)
   d. Let f be the result of calling the Object.[[Get]] method with base as the this object and argument P
   e. If f == undefined and Type(base) is XMLList and base.[[Length]] == 1
      i. Let r0 be a new Reference with base object = base[0] and property name = P
      ii. Return the result of calling CallMethod(r0, args) recursively
   f. If f == undefined and Type(base) is XML and base.hasSimpleContent () == true
      i. Let r0 be a new Reference with base object = ToObject(ToString(base)) and property name = P
      ii. Return the result of calling CallMethod(r0, args) recursively
4. If Type(f) is not an Object, throw a TypeError exception
5. If f does not implement the internal [[Call]] method, throw a TypeError exception
6. If base is an activation object, base = null
7. Return the result of calling the [[Call]] method of f providing base as the this value and the list args as the argument values

10.2.3 XML Descendant Accessor

Overview

E4X extends ECMAScript by adding a descendant accessor. The following productions describe the syntax of the descendant accessor:

MemberExpression :
  MemberExpression .. Identifier
  MemberExpression .. PropertyIdentifier

CallExpression :
  CallExpression .. Identifier
  CallExpression .. PropertyIdentifier

Overview

When the MemberExpression or CallExpression evaluate to an XML object or an XMLList, the descendant accessor examines all of the descendant XML properties (i.e., children, grand children, great-grandchildren, etc) of its left operand and returns an XMLList containing those with names that match its right operand in order. For example,
```javascript
var e = <employees>
  <employee id="1"><name>Joe</name><age>20</age></employee>
  <employee id="2"><name>Sue</name><age>30</age></employee>
</employees>;

var names = e..name;  // get all the names in e
```

**Semantics**

The production `MemberExpression : MemberExpression .. Identifier` is evaluated as follows:

1. Let `ref` be the result of evaluating `MemberExpression`
2. Let `x = GetValue(ref)`
3. If `Type(x) ∉ {XML, XMLList}`, throw a `TypeError` exception
4. Let `P` be a string value containing the same sequence of characters as `Identifier`
5. Return the result of calling the `[[Descendants]]` method of `x` with argument `P`

The production `CallExpression : CallExpression .. Identifier` is evaluated in exactly the same manner, except that the contained `CallExpression` is evaluated in step 1.

The production `MemberExpression : MemberExpression .. PropertyIdentifier` is evaluated as follows:

1. Let `oRef` be the result of evaluating `MemberExpression`
2. Let `x = GetValue(ref)`
3. If `Type(x) ∉ {XML, XMLList}`, throw a `TypeError` exception
4. Let `P` be the result of evaluating `PropertyIdentifier`
5. Return the result of calling the `[[Descendants]]` method of `x` with argument `P`

The production `CallExpression : CallExpression .. PropertyIdentifier` is evaluated in exactly the same manner, except that the contained `CallExpression` is evaluated in step 1.

### 10.2.4 XML Filtering Predicate Operator

**Syntax**

E4X extends ECMAScript by adding a filtering predicate operator. The following productions describe the syntax of the filtering predicate operator:

```
MemberExpression :
  MemberExpression . ( Expression )

CallExpression :
  CallExpression . ( Expression )
```

**Overview**

When the left operand evaluates to an XML object, the filtering predicate adds the left operand to the front of the scope chain of the current execution context, evaluates the `Expression` with the augmented scope chain, converts the result to a Boolean value, then restores the scope chain. If the result is `true`, the filtering predicate returns an `XMLList` containing the left operand. Otherwise it returns an empty `XMLList`.

When the left operand is an `XMLList`, the filtering predicate is applied to each XML object in the `XMLList` in order using the XML object as the left operand and the `Expression` as the right operand. It concatenates the results and returns them as a single `XMLList` containing all the XML properties for which the result was `true`. For example,
var john = e.employee.(name == "John");  // employees with name John
var twoemployees = e.employee.(@id == 0 || @id == 1);  // employees with id's 0 & 1
var emp = e.employee.(@id == 1).name;  // name of employee with id 1

The effect of the filtering predicate is similar to SQL’s WHERE clause or XPath’s filtering predicates.

For example, the statement:

// get the two employees with ids 0 and 1 using a predicate
var twoEmployees = e.employee.(@id == 0 || @id == 1);

produces the same result as the following set of statements:

// get the two employees with the ids 0 and 1 using a for loop
var i = 0;
var twoEmployees = new XMLList();
for (var p in e.employee) {
    if (p.@id == 0 || p.@id == 1) {
        twoEmployees[i++] = p;
    }
}

Semantics

The production MemberExpression : MemberExpression . ( Expression ) is evaluated as follows:

1. Let ref be the result of evaluating MemberExpression
2. Let x = GetValue(ref)
3. If Type(x) ∉ {XML, XMLList}, throw a TypeError exception
4. Let l = ToXMLList(x)
5. Let r be a new XMLList with r.([TargetObject]) = null
6. For i = 0 to l.([Length])-1
   a. Add x[i] to the front of the scope chain
   b. Let ref be the result of evaluating E using the augmented scope chain of step 6a
   c. Let match = ToBoolean(GetValue(ref))
   d. Remove x[i] from the front of the scope chain
   e. If (match == true), call the [[Append]] method of r with argument x[i]
7. Return r

The production CallExpression : CallExpression . ( Expression ) is evaluated in exactly the same manner, except that the contained CallExpression is evaluated in step 1.

10.3 Unary Operators

10.3.1 The Delete Operator

This section is provided to describe the effects of the XML [[Delete]] operators on the delete operator. E4X does not define any extensions to the syntax or semantics of the ECMAScript delete operator beyond those specified by the XML and XMLList [[Delete]] operators.

Syntax

E4X reuses the ECMAScript delete operator for deleting XML properties and XML attributes from XML objects and XMLLists. The syntax of the delete operator is described by the following production:

UnaryExpression :
delete UnaryExpression

Overview

When UnaryExpression evaluates to a Reference \( r \) with a base object of type XML, the delete operator removes the XML attributes or properties specified by the property name of \( r \) from the base object. When UnaryExpression evaluates to a Reference \( r \) with a base object of type XMLList, the delete operator removes the XML objects specified by the property name of \( r \) from the base object and the associated XML object. For example,

```
delete order.customer.address;       // delete the customer address
delete order.customer.@id;          // delete the customer ID
delete order.item.price[0];         // delete the first item price
delete order.item;                  // delete all the items
```

Semantics

E4X extends the semantics of the delete operator by providing more elaborate \([[\text{Delete}]]\) methods used when UnaryExpression evaluates to a value of type XML or XMLList (see sections 8.1.1.3 and 8.2.1.3 respectively).

10.3.2 The typeof Operator

Syntax

E4X reuses the syntax of the ECMAScript’s typeof operator for determining the types of XML and XMLList objects. The ECMAScript syntax for the typeof operator is described by the following production:

```
UnaryExpression : typeof UnaryExpression
```

Overview

E4X extends the semantics of the ECMAScript typeof operator for determining the types of XML and XMLList objects. When UnaryExpression evaluates to a value of type XML, the typeof operator returns the string "xml". When UnaryExpression evaluates to a value of type XMLList, the typeof operator returns the string "xmlList".

Semantics

The production UnaryExpression : typeof UnaryExpression is evaluated as follows:

1. Let \( u \) be the result of evaluating UnaryExpression
2. If Type(\( u \)) is Reference and GetBase(\( u \)) is null, return "undefined"
3. Return a string determined by Type(GetValue(\( u \))) according to the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>&quot;undefined&quot;</td>
</tr>
<tr>
<td>Null</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Boolean</td>
<td>&quot;boolean&quot;</td>
</tr>
<tr>
<td>Number</td>
<td>&quot;number&quot;</td>
</tr>
<tr>
<td>String</td>
<td>&quot;string&quot;</td>
</tr>
<tr>
<td>XML</td>
<td>&quot;xml&quot;</td>
</tr>
<tr>
<td>XMLList</td>
<td>&quot;xmlList&quot;</td>
</tr>
<tr>
<td>Object (native and doesn't implement [[Call]])</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Object (native and implements [[Call]])</td>
<td>&quot;function&quot;</td>
</tr>
</tbody>
</table>
## 10.4 Additive Operators

### Syntax

E4X reuses the syntax of the ECMAScript addition operator for concatenating two values of type XML or XMLList. The ECMAScript syntax for the addition operator is described by the following production:

```
AdditiveExpression : AdditiveExpression + MultiplicativeExpression
```

### 10.4.1 The Addition Operator (+)

#### Overview

E4X extends the semantics of the ECMAScript addition operator to perform either string concatenation, XML and XMLList concatenation or numeric addition depending on its arguments.

When both `AdditiveExpression` and `MultiplicativeExpression` evaluate to either an XML object or an XMLList, the addition operator starts by creating a new, empty XMLList as the return value. If the left operand evaluates to an XML object, it is added to the return value. If the left operand evaluates to an XMLList, each XML property of the XMLList is added to the return value in order. Likewise, if the right operand evaluates to an XML object, it is added to the return value. Otherwise, if it is an XMLList each XML property of the XMLList is added to the return value in order.

For example,

```javascript
// create an XMLList containing the elements <name>, <age> and <hobby>
var employeedata = <name>Fred</name> + <age>28</age> + <hobby>skiing</hobby>;

// create an XMLList containing three item elements extracted from the order element
var myitems = order.item[0] + order.item[2] + order.item[3];

// create a new XMLList containing all the items in the order plus one new one
var newitems = order.item + <item><description>new item</description></item>;
```

Note: Using the addition operator with operands of type XML and XMLList always results in an XMLList. When numeric addition of XML objects is desired, the operands must be explicitly coerced to Numbers. This may be accomplished by using the unary `+` operator or the Number conversion function. For example,

```javascript
// add the prices of the first and third items in the order (coersion with unary +)
var totalPrice = +order.item[0].price + +order.item[2].price

// add the prices of the second and fourth items in the order (coersion using Number conversion function)
var totalPrice = Number(order.item[1].price) + Number(order.item[3].price)
```

Likewise, when string concatenation of XML objects is desired, at least one of the operands must be explicitly coerced to a String. This may be accomplished by concatenating them to the empty string (""") or using the String conversion function. For example,

```javascript
// concatenate the street and the city of the customer's address (coersion with the empty string)
var streetcity = "" + order.customer.address.street + order.customer.address.city;

// concatenate the state and the zip of the customer's address (coersion using String conversion function)
var statezip = String(order.customer.address.state) + order.customer.address.zip;
```

### Semantics

The production `AdditiveExpression : AdditiveExpression + MultiplicativeExpression` is evaluated as follows:
1. Let \( a \) be the result of evaluating `AdditiveExpression`
2. Let \( left = \text{GetValue}(a) \)
3. Let \( m \) be the result of evaluating `MultiplicativeExpression`
4. Let \( right = \text{GetValue}(m) \)
5. If \( \text{Type}(left) \in \{\text{XML, XMLList}\} \) and \( \text{Type}(right) \in \{\text{XML, XMLList}\} \)
   a. Let \( l \) be a new XMLList
   b. Call the `[[Append]]` method of \( l \) with argument \( x \)
   c. Call the `[[Append]]` method of \( l \) with argument \( y \)
   d. Return \( l \)
6. Let \( pLeft = \text{ToPrimitive}(left) \)
7. Let \( pRight = \text{ToPrimitive}(right) \)
8. If \( \text{Type}(pLeft) \) is String or \( \text{Type}(pRight) \) is String
   a. Return the result of concatenating \( \text{ToString}(pLeft) \) and \( \text{ToString}(pRight) \)
9. Else
   a. Apply the addition operation to \( \text{ToNumber}(pLeft) \) and \( \text{ToNumber}(pRight) \) and return the result. See ECMAScript Edition 3, section 11.6.3 for details.

### 10.5 Equality Operators

#### 10.5.1 The Abstract Equality Comparison Algorithm

**Overview**

E4X extends the abstract equality comparison algorithm defined by ECMAScript to enable equality comparisons involving QName and Namespace objects and the types XML and XMLList.

**Semantics**

The comparison \( x \equiv y \), where \( x \) and \( y \) are values, produces \textit{true} or \textit{false}. This comparison is performed using the following steps:

1. If \( \text{Type}(x) \) is XMLList, call the `[[Equals]]` method of \( x \) with argument \( y \) and return the result
2. If \( \text{Type}(y) \) is XMLList, call the `[[Equals]]` method of \( y \) with argument \( x \) and return the result
3. If \( \text{Type}(x) \) is the same as \( \text{Type}(y) \)
   a. If \( \text{Type}(x) \) is XML
      i. If \( \text{x.hasSimpleContent}() \equiv \text{true} \) and \( \text{y.hasSimpleContent}() \equiv \text{true} \)
         1. Return the result of the comparison \( \text{ToString}(x) \equiv \text{ToString}(y) \)
   b. If \( \text{Type}(x) \) is Object and \( x.\text{[[Class]]} \equiv \text{"QName"} \)
      i. If the result of the comparison \( \text{x.uri} \equiv \text{y.uri} \) is \text{true} and the result of the comparison \( \text{x.localName} \equiv \text{y.localName} \) is \text{true}, return \text{true}. Otherwise, return \text{false}.
   c. If \( \text{Type}(x) \) is Object and \( x.\text{[[Class]]} \equiv \text{"Namespace"} \), return the results of the comparison \( \text{x.uri} \equiv \text{y.uri} \)
   d. If \( \text{Type}(x) \) is \text{undefined}, return \text{true}.
   e. If \( \text{Type}(x) \) is \text{null}, return \text{true}.
   f. If \( \text{Type}(x) \) is Number
      i. If \( x \) is \text{NaN}, return \text{false}.
      ii. If \( y \) is \text{NaN}, return \text{false}.
      iii. If \( x \) is the same number value as \( y \), return \text{true}.
      iv. If \( x \) is \( +0 \) and \( y \) is \( -0 \), return \text{true}.
      v. If \( x \) is \( -0 \) and \( y \) is \( +0 \), return \text{true}.
      vi. Return \text{false}.
   g. If \( \text{Type}(x) \) is String, then return \text{true} if \( x \) and \( y \) are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, return \text{false}.
   h. If \( \text{Type}(x) \) is boolean, return \text{true} if \( x \) and \( y \) are both \text{true} or both \text{false}. Otherwise, return \text{false}.
      i. Return \text{true} if \( x \) and \( y \) refer to the same object or if they refer to objects joined to each other (ECMAScript Edition 3 Section 13.1.2). Otherwise, return \text{false}.
4. If \( \text{Type}(x) \) is XML and \( x.\text{hasSimpleContent}() \equiv \text{true} \) or \( \text{Type}(y) \) is XML and \( y.\text{hasSimpleContent}() \equiv \text{true} \)
   a. Return the result of the comparison \( \text{ToString}(x) \equiv \text{ToString}(y) \)
5. If `x` is `null` and `y` is `undefined`, return `true`.
6. If `x` is `undefined` and `y` is `null`, return `true`.
7. If `Type(x)` is `Number` and `Type(y)` is `String`, return the result of the comparison `x == ToNumber(y)`.
8. If `Type(x)` is `String` and `Type(y)` is `Number`, return the result of the comparison `ToNumber(x) == y`.
9. If `Type(x)` is `Boolean`, return the result of the comparison `ToNumber(x) == y`.
10. If `Type(y)` is `Boolean`, return the result of the comparison `x == ToNumber(y)`.
11. If `Type(x)` is either `String` or `Number` and `Type(y)` is `Object`, return the result of the comparison `x == ToPrimitive(y)`.
12. If `Type(x)` is `Object` and `Type(y)` is either `String` or `Number`, return the result of the comparison `ToPrimitive(x) == y`.
13. Return `false`.

10.6 Assignment Operators

10.6.1 XML Assignment Operator

This section is provided to describe the effects of the XML `[[Put]]` operator on the assignment operator. E4X does not define any extensions to the syntax or semantics of the ECMAScript assignment operator beyond those specified by the XML and XMLList `[[Put]]` operators.

Syntax

E4X reuses the ECMAScript assignment operator to modify, replace and insert properties and XML attributes in an XML object. The ECMAScript syntax for the assignment operator is described by the following production:

```
AssignmentExpression : 
  LeftHandSideExpression = AssignmentExpression
```

Overview

The assignment operator begins by evaluating the `LeftHandSideExpression`, which resolves to a reference `r` consisting of a base object `parent` and a property name. If `parent` is an XML object, the assignment operator performs the steps described in section (see section 10.6.2 for the steps performed if `parent` is an XMLList).

If the `property name` begins with the character “@”, the XML assignment operator creates or modifies an XML attribute in the `parent`. If the named XML attribute already exists, the assignment operator modifies its value, otherwise it creates a new XML attribute with the given name and value. If `AssignmentExpression` evaluates to an XMLList, the value of the named attribute will be a space separated list of values (i.e., an XML attribute list) constructed by converting each value in the XMLList to a string and concatenating the results separated by spaces. If the `AssignmentExpression` does not evaluate to an XMLList, the value of the named attribute will be derived by evaluating the `AssignmentExpression` and calling `ToString` on the result. For example,

```javascript
order.item[1].@id = 123; // change the value of the id attribute on the second item
order.item[1].@newattr = "new value"; // add a new attribute to the second item
order.@allids = order.item.@id; // construct an attribute list containing all the ids in this order
```

If the `property name` is an array index, the XML assignment operator replaces an existing property or appends a new property to an XML object according to the property’s ordinal position within the XML object (i.e., its numeric property name). If a property already exists at the given location, the assignment operator replaces it, otherwise it appends a new property to the end of the `parent`. If the `AssignmentExpression` evaluates to an XML object, the assignment operator replaces the value of the property at the given position with a deep copy of the given XML object. If the `AssignmentExpression` evaluates to an XMLList, the assignment operator replaces the value of the property at the given position with a deep copy of each item in the XMLList in order, effectively deleting the original property and inserting the contents of the XMLList in its place. If the `AssignmentExpression` does not evaluate to a value of type XML or XMLList, the assignment operator calls `ToString` on the given value and replaces the property at the given position with the result. For example,
// replace the first child of the order element with an XML value
order[0] = <customer>
     <name>Fred</name>
     <address> ... </address>
</customer>;

// replace the second child of the order element with a list of items
order[1] = <item> item one </item>
   + <item> item two </item>
   + <item> item three </item>;

// replace the third child or the order with a text node
order[2] = "A text node";

// append a new item to the end of the order
order[order.length] = <item> new item </item>;

If the property name does not begin with "@" and is not an array index, the XML assignment operator replaces, modifies or appends one or more XML objects in the parent by XML name. If only one XML object exists with the given name and the AssignmentExpression evaluates to an XML object or XMLList, the assignment operator replaces the identified XML object with the given value. If there are no XML properties with the given name, a new XML property with the given name and value is appended to the end of the parent. If more than one XML object exists with the given name and the AssignmentExpression evaluates to an XML object or XMLList, the assignment operator replaces the first XML property with a matching name with the given value and deletes the remaining XML properties with the given name, essentially replacing all the XML object with the given name with the given value. If the AssignmentExpression does not evaluate to a XML object or XMLList, the assignment operator calls ToString on the given value and replaces the properties (i.e., the content) of the appropriate XML object (as opposed to replacing the XML object itself). This provides a simple, intuitive syntax for setting the value of a named XML property to a primitive value. For example,

item.price = 99.95;       // change the price of the item
item.description = "Mobile Phone"; // change the description of the item

Semantics

E4X extends the semantics of the assignment operator by providing more elaborate [[Put]] methods used when MemberExpression evaluates to a value of type XML or XMLList (see sections 8.1.1.2 and 8.2.1.2 respectively).

10.6.2 XMLList Assignment Operator

This section is provided to describe the effects of the XMLList [[Put]] operator on the assignment operator. E4X does not define any extensions to the syntax or semantics of the ECMAScript assignment operator beyond those provided by the XML and XMLList [[Put]] operators.

Syntax

E4X reuses the ECMAScript assignment operator to replace or append values to XMLLists and their associated XML objects. The ECMAScript syntax for the assignment operator is described by the following production:

\[
\text{AssignmentExpression} : \\
\text{LeftHandSideExpression} = \text{AssignmentExpression}
\]

Overview

The assignment operator begins by evaluating the LeftHandSideExpression, which resolves to a reference \( r \) consisting of a base object \( \text{parent} \) and a property name. If \( \text{parent} \) is an XMLList, the assignment operator performs the steps described in this section (see section 10.6.1 for the steps performed when \( \text{parent} \) is an XML object).
If the property name is not an array index, the XMLList assignment operator checks to see if this XMLList object contains only 1 item and that item is of type XML. If so, the XMLList assignment operator delegates its behaviour to the [[Put]] method of the XML object it contains (see section 8.1.1.2). Otherwise, it throws a TypeError exception. This treatment intentionally blurs the distinction between a single XML object and an XMLList containing only one XML object. For example,

```
// set the name of the only customer in the order to Fred Jones
order.customer.name = "Fred Jones";

// replace all the hobbies for the only customer in the order
order.customer.hobby = "shopping";

// attempt to set the sale date of the item. Throw an exception if more than 1 item exists.
order.item.saledate = "05-07-2002";

// replace all the employee's hobbies with their new favorite pastime
emps.employee.(@id == 3).hobby = "working";
```

In the first statement above, the expression “order.customer” returns an XMLList containing only one XML item. The expression “order.customer.name” implicitly converts this XMLList to an XML value and assigns the value “Fred Jones” to that value.

If the property name is an array index, the assignment operator replaces the property identified by property name in the XMLList or appends a new property if none exists with that property name. In addition, if the property identified is an XML value with a non-null parent, the XML value is also replaced in the context of its parent. If the AssignmentExpression evaluates to an XML value, the assignment operator replaces the value of the property identified by property name with a deep copy of the given XML object. If the AssignmentExpression evaluates to an XMLList, the assignment operator replaces the value of the property identified by property name with a deep copy of each item in the XMLList in order, effectively deleting the original property and inserting the contents of the XMLList in its place. If the AssignmentExpression does not evaluate to a value of type XML or XMLList, the assignment operator calls ToString on the given value and replaces the property at the given position with the result. Here are some examples,

```
// replace the first employee with George
e.employee[0] = <employee><name>George</name><age>27</age></employee>;

// add a new employee to the end of the employee list
e.employee[e.employee.length] = <employee><name>Frank</name></employee>;
```

Semantics

E4X extends the semantics of the assignment operator by providing more elaborate [[Put]] methods used when MemberExpression evaluates to a value of type XML or XMLList (see sections 8.1.1.2 and 8.2.1.2 respectively).

10.6.3 Compound Assignment (op=)

This section is provided to describe the effects of the XML and XMLList [[Get]] and [[Put]] operators on the compound assignment operator. E4X does not define any extensions to the syntax or semantics of the ECMAScript compound assignment operator beyond those provided by the XML and XMLList [[Get]] and [[Put]] operators.

Syntax

E4X benefits from the compound assignment operator “+=” without requiring additional ECMAScript extensions. The syntax of the compound assignment “+=” is described by the following production:

```
AssignmentExpression : LeftHandSideExpression += AssignmentExpression
```
Overview

When the left operand is an XML object, the “+=” operator has the effect of inserting one or more XML elements specified by the right operand just after the ordinal position of the left operand within its parent. For example,

```javascript
var e = <employees>
   <employee id="1"><name>Joe</name><age>20</age></employee>
   <employee id="2"><name>Sue</name><age>30</age></employee>
</employees>;

// insert employee 3 and 4 after the first employee
e.employee[0] += <employee id="3"><name>Fred</name></employee> +
   <employee id="4"><name>Carol</name></employee>;
```

Following the expressions above, the variable “e” would contain the XML value:

```xml
<employees>
   <employee id="1"><name>Joe</name><age>20</age></employee>
   <employee id="3"><name>Fred</name></employee>
   <employee id="4"><name>Carol</name></employee>
   <employee id="2"><name>Sue</name><age>30</age></employee>
</employees>;
```

When the left operand is an XMLList, the “+=” operator has the effect of appending one or more values specified by the right operand to the XMLList. If the last item in the XMLList is an XML object with a non-null parent, the “+=” operator also appends the items to the XML object referred to by parent just after the position of the last item in the XMLList. For example,

```javascript
var e = <employees>
   <employee id="1"><name>Joe</name><age>20</age></employee>
   <employee id="2"><name>Sue</name><age>30</age></employee>
</employees>;

// append employees 3 and 4 to the end of the employee list
e.employee += <employee id="3"><name>Fred</name></employee> +
   <employee id="4"><name>Carol</name></employee>;
```

Following the expressions above, the variable “e” would contain the XML value:

```xml
<employees>
   <employee id="1"><name>Joe</name><age>20</age></employee>
   <employee id="2"><name>Sue</name><age>30</age></employee>
   <employee id="3"><name>Fred</name></employee>
   <employee id="4"><name>Carol</name></employee>
</employees>;
```

Semantics

E4X extends the semantics of the compound assignment operator by providing more elaborate [[Get]] and [[Put]] methods used when MemberExpression evaluates to a value of type XML or XMLList (see sections 8.1.1.1, 8.1.1.2, 8.2.1.1 and 8.2.1.2 respectively).
11 Statements
E4X extends the statements provided in ECMAScript with the following production:

Statement :
  DefaultXMLNamespaceStatement

11.1 Default XML Namespace Statement
Syntax
E4X extends ECMAScript by adding a default xml namespace statement. The following production describes the syntax of the default xml namespace statement:

DefaultXMLNamespaceStatement :
  default xml namespace = Expression

Overview
The default xml namespace statement sets the value of the internal property \[[DefaultNamespace]\] of the variable object associated with the current execution context (see section 10 of ECMAScript Edition 3). If the variable object associated with the current execution context does not have an internal property \[[DefaultNamespace]\], the default xml namespace statement creates one. If the default xml namespace statement occurs inside a FunctionDeclaration, the default xml namespace is defined with function-local scope in that function. Otherwise, the default xml namespace is defined with global scope. Initially, the default xml namespace of the global scope is set to no namespace (section 12.1.1.1).

When the default xml namespace statement is executed, it evaluates the Expression, converts the result to a String s, creates a new Namespace object n as if by calling the constructor \( n = \text{new} \text{Namespace}('', s) \), and sets the default XML namespace associated with the current execution context to n. Unqualified XML element names following the default xml namespace declaration in the current scope will be associated with the default xml namespace specified by Expression. For example,
// declare some namespaces and a default namespace for the current scope
var soap = new Namespace("http://schemas.xmlsoap.org/soap/envelope/");
var stock = new Namespace("http://mycompany.com/stocks");
default xml namespace = soap; // alternately, may specify full URI

// Create an XML initializer in the default (i.e., soap) namespace
var message = <Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <Body>
        <m:GetLastTradePrice xmlns:m="http://mycompany.com/stocks">
            <symbol>DIS</symbol>
        </m:GetLastTradePrice>
    </Body>
</Envelope>

// extract the soap encoding style using a QualifiedIdentifier (unqualified attributes are in no namespace)
var encodingStyle = message.@soap::encodingStyle;

//extract the body from the soap message using the default namespace
var body = message.Body;

// change the stock symbol using the default namespace and qualified names
message.Body.stock::GetLastTradePrice.stock::symbol = "MYCO";

Semantics

The production DefaultXMLNamespaceStatement : default xml namespace = Expression is evaluated as follows:

1. Let uriRef be the result of evaluating Expression
2. Let uriString = ToString(GetValue(uriRef))
3. Let namespace be a new Namespace Object, created as if by calling the constructor new Namespace("");
4. Let varObj be the variable object associated with the current execution context (see section 10.1.3 of ECMAScript Edition 3)
5. Let varObj.[[DefaultNamespace]] = namespace

The value of the default xml namespace is obtained using the internal operator GetDefaultNamespace. When the internal GetDefaultNamespace method is called, the following steps are taken:

1. While (there are more objects on the scope chain)
   a. Let o be the next object on the scope chain
   b. If o has the internal property [[DefaultNamespace]], return o.[[DefaultNamespace]]

11.2 The for-in Statement

Syntax

E4X extends the semantics of the ECMA-Script for-in statements for iterating through the properties of values of type XML and XMLList. The syntax of the for-in statements is specified by the following productions:

IterationStatement :
  for ( LeftHandSideExpression in Expression ) Statement
  for ( var VariableDeclarationNoLn in Expression ) Statement
Overview

When the value of Expression evaluates to a value of type XML or XMLList, the for-in statements iterate through each property in the resulting value in order. For each property, the for-in operators assign the value of the property to the variable identified by LeftHandSideExpression or VariableDeclarationNoLn and evaluate the Statement. For example:

```javascript
// print all the employee names
for (var n in e..name) {
    print ("Employee name: " + n);
}

// print each child of the first item
for (var child in order.item[0]) {
    print("item child: " + child);
}
```

In the first for-in statement above, the expression “e..name” returns an XMLList containing all of the descendant XML properties of the XML object “e” with the name “name”. The for-in statement iterates through the list in order. For each XML property in the list, it assigns the value of the XML property to the variable “n” and executes the code nested in curly braces. Similarly, in the second for-in statement above, the expression “order.item[0]” returns the first XML object named “item” from the XML object named “order”. The for-in statement iterates through each property of the XML object in order assigning the value of the XML property to the variable “child” and executing the code nested in curly braces.

Note: The for-in operators behave differently for values of type XML and XMLList than they do for native ECMAScript objects. With native ECMAScript objects, for-in assigns the loop variable over the domain of the array. However with values of type XML or XMLList, for-in assigns the loop variable over the range of the array.

Semantics

The production IterationStatement : for ( LeftHandSideExpression in Expression ) Statement is evaluated as follows:

1. Let ref be the result of evaluating Expression
2. Let e = GetValue(ref)
3. if Type(e) ∈ {XML, XMLList}
   a. Let l be a shallow copy of e
4. else
   a. Let l = ToObject(e)
5. Let V = empty
6. While (l has more properties)
   a. Let p be the next property of l (see notes below)
   b. If p does not have the DontEnum attribute
      i. Let i be the result of evaluating LeftHandSideExpression
      ii. If Type(l) ∈ {XML, XMLList}
          1. Let value be the value of property p
          2. PutValue(i, value)
      iii. Else
          1. Let name be the name of property p
          2. PutValue(i, name)
      iv. Let s be the result of evaluating Statement
      v. If s.value is not empty, let V = s.value
      vi. If s.type is break and s.target is in the current label set, return (normal, V, empty)
      vii. If s.type is not continue or (s.target is not in the current label set)
          1. If s is an abrupt completion, return s
7. Return (normal, V, empty)

The production IterationStatement : for ( var VariableDeclarationNoLn in Expression ) Statement is evaluated as follows:
1. Let varRef be the result of evaluating \textit{VariableDeclarationNoLn}
2. Let \( r \) ef be the result of evaluating \textit{Expression}
3. Let \( e = \text{GetValue}(\text{ref}) \)
4. if Type(\( e \)) \( \in \{ \text{XML, XMLList} \) 
   a. Let \( l \) be a shallow copy of \( e \)
5. else
   a. Let \( l = \text{ToObject}(e) \)
6. Let \( V = \text{empty} \)
7. While \( l \) has more properties
   a. Let \( p \) be the next property of \( l \) (see notes below)
   b. If \( p \) does not have the \textit{DontEnum} attribute
      i. Let \( i \) be the result of evaluating \textit{varRef} as if it were an identifier (See section 11.1.2 of ECMAScript Edition 3)
      ii. If Type(\( l \)) \( \in \{ \text{XML, XMLList} \) 
         1. Let \( \text{value} \) be the value of property \( p \)
         2. PutValue(\( i, \text{value} \) )
      iii. Else
         1. Let \( \text{name} \) be the name of property \( p \)
         2. PutValue(\( i, \text{name} \) )
   iv. Let \( s \) be the result of evaluating \textit{Statement}
   v. If \( s\).value is \textit{not empty}, let \( V = s\).value
   vi. If \( s\).type is \textit{break} and \( s\).target is in the current label set, return (\textit{normal}, \( V, \text{empty} \) )
   vii. If \( s\).type is \textit{not continue} or \( s\).target is not in the current label set
      1. If \( s \) is an abrupt completion, return \( s \)
8. Return (\textit{normal}, \( V, \text{empty} \) )

When \( e \) does not evaluate to a value of type XML or XMLList, the mechanics of enumerating the properties (steps 6 and 6a in the first algorithm, steps 7 and 7a in the second) is implementation dependent. The order of enumeration is defined by the object. Properties of the object being enumerated may be deleted during enumeration. If a property that has not yet been visited during enumeration is deleted, then it will not be visited. If new properties are added to the object being enumerated during enumeration, the newly added properties are not guaranteed to be visited in the active enumeration. Enumerating the properties of an object includes enumerating properties of its prototype and the prototype of the prototype, and so on, recursively; but a property of a prototype is not enumerated if it is “shadowed” because some previous object in the prototype chain has a property with the same name.

When \( e \) evaluates to a value of type XML or XMLList, properties are enumerated in document order (steps 6 and 6a in the first algorithm and steps 7 and 7a in the second algorithm). Properties of the object \( e \) may be deleted, added or moved during enumeration. However, because the algorithms construct a copy of the properties of \( e \) prior to enumeration (step 3a in the first algorithm and 4a in the second algorithm), each property that is in \( e \) prior to enumeration will be visited once and only once regardless of mutations that occur to \( e \) during enumeration.
12 Native E4X Objects
E4X adds two native objects to ECMAScript, the native XML object and the native XMLList object. In addition, E4X adds new properties to the global object.

12.1 The Global Object

12.1.1 Internal Properties of the Global Object
E4X extends ECMAScript by adding the following internal properties to the global object.

12.1.1.1 [[DefaultNamespace]]
Overview
Initially, the global object has an internal property [[DefaultNamespace]] with its value set to a Namespace object representing no namespace, created as if by calling the Namespace constructor with no arguments. Consequently, unless otherwise specified using the default xml namespace statement (see section 11.1), unqualified names used to specify properties of XML objects will match XML properties in no namespace.

12.1.2 Function Properties of the Global Object
E4X extends ECMAScript by adding the following function properties to the global object.

12.1.2.1 isXMLName (value)
Overview
The isXMLName function examines the given value and determines whether it is a valid XML name that can be used as an XML element or attribute name. If so, it returns true, otherwise it returns false.

Semantics
When the isXMLName function is called with one parameter value, the following steps are taken:

1. Let q be a new QName created as if by calling the constructor new QName (value) and if a TypeError exception is thrown, return false
2. If q.localName does not match the production NCName, return false
3. Return true

Where the production NCName is defined in Section 2 of the Namespaces in XML specification.

12.1.3 Constructor Properties of the Global Object
E4X extends ECMAScript by adding the following constructor properties to ECMAScript.

12.1.3.1 Namespace ( . . . )
See section 12.2.1 and 12.2.

12.1.3.2 QName ( . . . )
See section 12.3.1 and 12.3.2.

12.1.3.3 XML ( . . . )
See sections 12.4.1 and 12.4.2.

12.1.3.4 XMLList ( . . . )
See section 12.5.1 and 1.
12.2 Namespace Objects

Namespace objects represent XML namespaces and provide an association between a namespace prefix and a Unique Resource Identifier (URI). The prefix is either the undefined value or a string value that may be used to reference the namespace within the lexical representation of an XML value. When an XML object containing a namespace with an undefined prefix is converted to a string, the implementation will automatically generate a prefix. The URI is a string value used to uniquely identify the namespace.

12.2.1 The Namespace Constructor Called as a Function

Syntax

\[
\text{Namespace ( )} \\
\text{Namespace ( uriValue )} \\
\text{Namespace ( prefixValue , uriValue )}
\]

Overview

If the Namespace constructor is called as a function with exactly one argument that is a Namespace object, the argument is returned unchanged. Otherwise, a new Namespace object is created and returned as if the same arguments were passed to the object creation expression \text{new Namespace ( … )}. See section 12.2.2.

Semantics

When Namespace is called as a function with no arguments, one argument \text{uriValue}, or two arguments \text{prefixValue} and \text{uriValue}, the following steps are taken:

1. If \text{prefixValue} is not specified and \text{Type(uriValue) is Namespace}, return \text{uriValue}
2. Create and return a new Namespace object exactly as if the Namespace constructor had been called with the same arguments (section 12.2.2).

12.2.2 The Namespace Constructor

Syntax

\[
\text{new Namespace ( )} \\
\text{new Namespace ( uriValue )} \\
\text{new Namespace ( prefixValue , uriValue )}
\]

Overview

When Namespace is called as part of a new expression, it is a constructor and creates a new Namespace object.

The [[Prototype]] property of the newly constructed object is set to the original Namespace prototype object, the one that is the initial value of Namespace.prototype (section 12.2.3.1). The [[Class]] property of the newly constructed object is set to “Namespace”.

When no arguments are specified, the namespace \text{uri} and the prefix are set to the empty string. A namespace with \text{uri} set to the empty string represents no namespace. No namespace is used in XML objects to explicitly specify that a name is not inside a namespace and may never be associated with a prefix other than the empty string.

When only the \text{uriValue} argument is specified and \text{uriValue} is a Namespace object, a copy of the \text{uriValue} is returned. When only the \text{uriValue} is specified and it is the empty string, the prefix is set to the empty string. In all other cases where only the \text{uriValue} is specified, the namespace prefix is set to the undefined value.

When the \text{prefixValue} argument is specified and set to the empty string, the Namespace is called a default namespace. Default namespaces are used in XML objects to implicitly specify the namespace of qualified names that do not specify a qualifier.

Semantics
When the Namespace constructor is called with no arguments, one argument `uriValue` or two arguments `prefixValue` and `uriValue`, the following steps are taken:

1. Create a new Namespace object `n`
2. If (`prefixValue` is not specified and `uriValue` is not specified)
   a. Let `n.prefix` be the empty string
   b. Let `n.uri` be the empty string
3. Else if (`prefixValue` is not specified)
   a. If (`uriValue` is `undefined`), throw a `TypeError` exception
   b. If (Type(`uriValue`) is Namespace)
      i. Let `n.prefix = uriValue.prefix`
      ii. Let `n.uri = uriValue.uri`
   c. Else
      i. Let `n.uri = ToString(`uriValue``)
      ii. If (`n.uri` is the empty string), let `n.prefix` be the empty string
      iii. Else let `n.prefix = undefined`
4. Else
   a. If (`uriValue` is `undefined`), throw a `TypeError` exception
   b. Let `n.uri = ToString(`uriValue``)
   c. If (`n.uri` is the empty string)
      i. If (`prefixValue` is `undefined` or `ToString`(`prefixValue`) is the empty string)
         1. Let `n.prefix` be the empty string
      ii. Else throw a `TypeError` exception
   d. Else if (`prefixValue` is `undefined`), let `n.prefix = undefined`
   e. Else if (isXMLName(`prefixValue`) == `false`)
      i. Let `n.prefix = undefined`
   f. Else let `n.prefix = ToString(`prefixValue``)
5. Return `n`

12.2.3 Properties of the Namespace Constructor
The value of the internal `[[Prototype]]` property of the Namespace constructor is the Function prototype object.

Besides the internal properties and the length property (whose value is 2), the Namespace constructor has the following properties.

12.2.3.1 Namespace.prototype
The initial value of the Namespace.prototype property is the Namespace prototype object (section 12.2.4).

This property has the attributes `{ DontEnum, DontDelete, ReadOnly }`.

12.2.4 Properties of the Namespace Prototype Object (Built-in Methods)
The Namespace prototype object is itself a Namespace object (its `[[Class]]` is “Namespace”) with its `uri` and `prefix` properties set to the empty string.

The value of the internal `[[Prototype]]` property of the Namespace prototype object is the Object prototype object (section 15.2.3.1 of ECMAScript Edition 3).

12.2.4.1 Namespace.prototype.constructor
The initial value of the Namespace.prototype.constructor is the built-in Namespace constructor.

12.2.4.2 Namespace.prototype.toString()
Overview

The `toString()` method returns a string representation of this Namespace object.

Semantics
When the toString method of a Namespace object \( n \) is called with no arguments, the following step is taken:

1. Return \( n.uri \)

### 12.2.5 Properties of Namespace Instances

Namespace instances inherit properties from the Namespace prototype object and also have a \textit{prefix} property and a \textit{uri} property.

#### 12.2.5.1 \textit{prefix}

The value of the \textit{prefix} property is either the \texttt{undefined} value or a string value. When the value of the \textit{prefix} property is the empty string, the Namespace is called a \textit{default namespace}. Default namespaces are used in XML objects to determine the namespace of names that do not specify a qualifier.

This property has the attributes \{\texttt{DontDelete}, \texttt{ReadOnly}\}

#### 12.2.5.2 \textit{uri}

The value of the \textit{uri} property is a string value. When the value of the \textit{uri} property is the empty string, the Namespace represents the \textit{unnamed namespace}. The unnamed namespace is used in XML objects to explicitly specify that a name is not inside a namespace.

This property has the attributes \{\texttt{DontDelete}, \texttt{ReadOnly}\}

### 12.3 QName Objects

The QName type is used to represent qualified names of XML elements and attributes. The QName type specifies a namespace URI and a local name. The local name must be a value of type string. The namespace URI must be a value of type string or \texttt{undefined}. When the namespace URI is \texttt{undefined}, this qualified name matches any namespace. In addition, implementations may include an internal \texttt{[[Prefix]]} property that is not directly visible to E4X users. The internal \texttt{[[Prefix]]} property may be used to preserve the prefix of the Namespace associated with the QName. If no namespace prefix was specified for the associated Namespace, the \texttt{[[Prefix]]} property may be \texttt{undefined}.

A value of type QName may be specified using a \texttt{QualifiedIdentifier}. If the QName of an XML element is specified without identifying a namespace (i.e., as an unqualified identifier), the \textit{uri} property of the associated QName will be the current in-scope default namespace (section 11.1). If the QName of an XML attribute is specified without identifying a namespace, the \textit{uri} property of the associated QName will be the null string representing no namespace.

#### 12.3.1 The QName Constructor Called as a Function

**Syntax**

\[
\text{QName} \ (\text{Name})
\]

\[
\text{QName} \ (\text{Namespace, Name})
\]

**Overview**

When QName is called as a function rather than as a constructor, it creates and initializes a new QName object. Thus the function call \texttt{QName} ( \ldots ) is equivalent to the object creation expression \texttt{new QName} ( \ldots ) with the same arguments. See section 12.3.2.

**Semantics**

When the QName function is called the following step is taken.

1. Create and return a new QName object exactly as if the QName constructor had been called with the same arguments (section 12.3.2).

#### 12.3.2 The QName Constructor

**Syntax**
new QName ( Name )
new QName ( Namespace , Name )

Overview

When QName is called as part of a new expression, it is a constructor and creates a new QName object.

The [[Prototype]] property of the newly constructed object is set to the original QName prototype object, the one that is the initial value of QName.prototype (section 12.3.3.1). The [[Class]] property of the newly constructed object is set to “QName”.

Name may be a QName or any value that may be converted to a String using ToString. If Name is a QName, the localName of the newly created QName will be equal to the localName of Name. If Name is a QName and Namespace is not specified, the QName constructor returns a copy of the given Name.

When both the Namespace and Name arguments are specified, the localName property of the newly created object is set according to the given Name and the uri property of the newly created object is set according to the Namespace argument. If the Namespace argument is a Namespace, the uri property of the newly created object is set to the uri property of the Namespace object. If the Namespace property is null, the uri property of the newly created object will be null, meaning it will match names in any namespace.

Semantics

When the QName constructor is called with a one argument Name or two arguments Namespace and Name the following steps are taken:

1. If (Type(Name) is Object and Name.[[Class]] == "QName")
   a. If (Namespace is not specified), return a copy of Name
   b. Else let Name = Name.localName
3. Let Name = ToString(Name)
4. If (Namespace is undefined or not specified)
   a. If Name = "*"
      1. Let Namespace = null
   b. Else
      i. Let Namespace = GetDefaultNamespace()
5. Let q be a new QName with q.localName = Name
6. If Namespace == null
   a. Let q.uri = null
      Note: implementations that preserve prefixes in qualified names may also set q.[[Prefix]] to undefined
7. Else
   a. Let Namespace be a new Namespace created as if by calling the constructor new Namespace(Namespace)
   b. Let q.uri = Namespace.uri
      Note: implementations that preserve prefixes in qualified names may also set q.[[Prefix]] to Namespace.prefix
8. Return q

12.3.3 Properties of the QName Constructor

The value of the internal [[Prototype]] property of the QName constructor is the Function prototype object.

Besides the internal properties and the length property (whose value is 2), the QName constructor has the following properties.

12.3.3.1 QName.prototype

The initial value of the QName.prototype property is the QName prototype object (section 12.3.4).

This property has the attributes { DontEnum, DontDelete, ReadOnly }.

12.3.4 Properties of the QName Prototype Object

The QName prototype object is itself a QName object (its [[Class]] is “QName”) with its uri and localName properties set to the empty string.
The value of the internal [[Prototype]] property of the QName prototype object is the Object prototype object (section 15.2.3.1 of ECMAScript Edition 3).

12.3.4.1 QName.prototype.constructor
The initial value of the QName.prototype.constructor is the built-in QName constructor.

12.3.4.2 QName.prototype.toString()

Overview

The toString method returns a string representation of this QName object.

Semantics

When the toString method of a QName object is called with no arguments, the following steps are taken:

1. Let s be the empty string
2. If n.uri is not the empty string
   a. If n.uri == undefined, let s be the string "*::"
   b. Else let s be the result of concatenating n.uri and the string "::"
3. Let s be the result of concatenating s and n.localName
4. Return s

12.3.5 Properties of QName Instances

QName instances inherit properties from the QName prototype object and also have a uri property, a localName property and an optional internal [[Prefix]] property that may be used by implementations that preserve prefixes in qualified names.

12.3.5.1 localName
The value of the localName property is a value of type string. When the value of the localName property is "*" it represents a wildcard that matches any name.

This property shall have the attributes { DontDelete, ReadOnly }

12.3.5.2 uri
The value of the uri property is a value of type string identifying the namespace of this QName. When the value of the uri property is the empty string, this QName is said to be in no namespace. No namespace is used in XML objects to explicitly specify that a name is not inside a namespace.

This property shall have the attributes { DontDelete, ReadOnly }

12.3.5.3 [[Prefix]]
The [[Prefix]] property is an optional internal property that is not directly visible to users. It may be used by implementations that preserve prefixes in qualified names. The value of the [[Prefix]] property is a value of type string or undefined. If the [[Prefix]] property is undefined, the prefix associated with this QName is unknown.

This property shall have the attributes { DontDelete, ReadOnly, DontEnum }

12.3.5.4 [[GetNamespace]] ( [ InScopeNamespaces ] )

Overview

The [[GetNamespace]] method is an internal method that returns a Namespace object with a URI matching the URI of this QName. InScopeNamespaces is an optional parameter. If InScopeNamespaces is unspecified, it is set to the empty set. If one or more Namespaces exists in InScopeNamespaces with a URI matching the URI of this QName, one of the matching Namespaces will be returned. If no such namespace exists in InScopeNamespaces, [[GetNamespace]] creates and returns a new Namespace with a URI matching that of this QName. For implementations that preserve prefixes in QNames, [[GetNamespace]] may return a Namespace that also has a matching prefix. The input variable InScopeNamespaces is a set of Namespace objects.
Semantics

When the [[GetNamespace]] method of a QName \( q \) is called with no arguments or one argument \( \text{InScopeNamespaces} \), the following steps are taken:

1. If \( \text{InScopeNamespaces} \) was not specified, let \( \text{InScopeNamespaces} = \{ \} \)
2. Find a Namespace \( ns \) in \( \text{InScopeNamespaces} \), such that \( ns\.uri = q\.uri \). If more than one such Namespace \( ns \) exists, the implementation may choose one of the matching Namespaces arbitrarily. Note: implementations that preserve prefixes in qualified names may additionally constrain \( ns \), such that \( ns\.prefix = q[[\text{Prefix}]] \)
3. If no such a namespace \( ns \) exists
   a. Let \( ns \) be a new namespace created as if by calling the constructor new Namespace(\( q\.uri \))
      Note: implementations that preserve prefixes and qualified names may create the new namespaces as if by calling the constructor Namespace(\( q.[[\text{Prefix}]], q\.uri \))
4. return \( ns \)

12.3.5.5 [[SetNamespace]] ( Namespace )

Overview

The [[SetNamespace]] method is an internal method that sets the URI of this QName according to the URI of the given Namespace. For implementations that preserve prefixes in QNames, it may also set the namespace prefix of this QName according to the given Namespace. The input variable Namespace may be a value of type Namespace.

Semantics

When the [[SetNamespace]] method of a QName \( q \) is called with namespace Namespace, the following steps are taken:

1. If Namespace.uri === undefined
   a. Let \( q\.uri \) be the empty string
      Note: implementations that preserve prefixes in qualified names may set \( q[[\text{Prefix}]] \) to undefined
2. Else
   a. Let \( q\.uri = \text{Namespace}.uri \)
      Note: implementations that preserve prefixes in qualified names may set \( q[[\text{Prefix}]] \) to Namespaces.prefix

12.4 XML Objects

12.4.1 The XML Constructor Called as a Function

Syntax

```
XML ( [ value ] )
```

Overview

When XML is called as a function rather than as a constructor, it performs a type conversion. If no argument is provided, the XML function returns an XML object representing an empty text node.

Semantics

When the XML function is called with no arguments or with one argument value, the following step is taken.

1. If value is \textbf{null}, undefined or not supplied, let value be the empty string
2. Return ToXML(value)
   Note: The ToXML operator is defined for implementations of the W3C information set. Implementations may expose this functionality to users via the XML constructor; however, this is not required. See section 9.3.2 for additional information.
12.4.2 The XML Constructor

Syntax

\[
\text{new XML (\{ value \})}
\]

Overview

When XML is called as part of a new expression, it is a constructor and may create a new XML object. When the XMLList constructor is called with no arguments, it returns an XML object representing an empty text node.

Semantics

When the XML constructor is called with no arguments or a single argument value, the following steps are taken:

1. If value is null, undefined or not supplied, let value be the empty string
2. Let \( x = \text{ToXML}(value) \)
   Note: The ToXML operator is defined for implementations of the W3C information set. Implementations may expose this functionality to users via the XML constructor; however, this is not required. See section 9.3.2 for additional information.
3. If Type(value) \( \in \{ \text{XML, XMLList, W3C XML Information Item} \} \)
   a. Return the result of calling the [[DeepCopy]] method of \( x \)
4. Return \( x \)

12.4.3 Properties of the XML Constructor

The value of the internal \([\text{Prototype}]\) property of the XML constructor is the Function prototype object.

Besides the internal properties and the length property (whose value is 1), the XML constructor has the following properties:

12.4.3.1 XML.prototype

The initial value of the XML.prototype property is the XML prototype object (section 12.4.3.7).

This property has the attributes \{ DontEnum, DontDelete, ReadOnly \}.

12.4.3.2 XML.ignoreComments

The initial value of the ignoreComments property is true. If ignoreComments is true, XML comments are ignored when constructing new XML objects. This property has the attributes \{ DontEnum, DontDelete \}.

12.4.3.3 XML.ignoreProcessingInstructions

The initial value of the ignoreProcessingInstructions property is true. If ignoreProcessingInstructions is true, XML processing instructions are ignored when constructing new XML objects. This property has the attributes \{ DontEnum, DontDelete \}.

12.4.3.4 XML.ignoreWhitespace

The initial value of the ignoreWhitespace property is true. If ignoreWhitespace is true, insignificant whitespace characters are ignored when processing constructing new XML objects. When elements tags and/or embedded expressions are separated only by whitespace characters, those whitespace characters are defined to be insignificant. Whitespace characters are defined to be space (\u0020), carriage return (\u000D), line feed (\u000A) and tab (\u0009). This property has the attributes \{ DontEnum, DontDelete \}.

12.4.3.5 XML.prettyPrinting

The initial value of the prettyPrinting property is true. If prettyPrinting is true, the ToString and ToXMLString operators will normalize whitespace characters between certain tags to achieve a uniform and aesthetic appearance. This property has the attributes \{ DontEnum, DontDelete \}.

12.4.3.6 XML.prettyIndent

The initial value of the prettyIndent property is 2. If the prettyPrinting property of the XML constructor is true, the ToString and ToXMLString operators will normalize whitespace characters between certain tags to achieve a uniform and aesthetic
appearance. Certain child nodes will be indented relative to their parent node by the number of spaces specified by prettyIndent. This property has the attributes {DontEnum, DontDelete}.

12.4.3.7 XML.settings ( )

Overview

The settings method is a convenience method for managing the collection of global XML settings stored as properties of the XML constructor (sections 12.4.3.2 through 12.4.3.6). It returns an object containing the properties of the XML constructor used for storing XML settings. This object may later be passed as an argument to the setSettings method to restore the associated settings. For example,

```javascript
// Create a general purpose function that may need to save and restore XML settings
function getXMLCommentsFromString(xmlString) {
    // save previous XML settings and make sure comments are not ignored
    var settings = XML.settings();
    XML.ignoreComments = false;

    var comments = XML(xmlString).comment();

    // restore settings and return result
    XML.setSettings(settings);
    return comments;
}
```

Semantics

When the settings method of the XML constructor is called, the following steps are taken:

1. Let `s` be a new Object created as if by calling the constructor `new Object()
2. Let `s.ignoreComments` = XML.ignoreComments
3. Let `s.ignoreProcessingInstructions` = XML.ignoreProcessingInstructions
4. Let `s.ignoreWhitespace` = XML.ignoreWhitespace
5. Let `s.prettyPrinting` = XML.prettyPrinting
6. Let `s.prettyIndent` = XML.prettyIndent
7. Return `s`

12.4.3.8 XML.setSettings ( [ Settings ] )

The setSettings method is a convenience method for managing the collection of global XML settings stored as properties of the XML constructor (sections 12.4.3.2 through 12.4.3.6). It may be used to restore a collection of XML settings captured earlier using the associated settings method. When called with a single argument `settings`, the setSettings method copies the properties of the XML constructor used for storing XML settings from the `settings` object. When called with no arguments, the setSettings method restores the default XML settings.

Semantics

When the setSettings method of the XML constructor is called with no arguments or with a single argument `settings`, the following steps are taken:

1. If `settings` is null, undefined or not provided
   a. Let XML.ignoreComments = true
   b. Let XML.ignoreProcessingInstructions = true
   c. Let XML.ignoreWhitespace = true
   d. Let XML.prettyPrinting = true
   e. Let XML.prettyIndent = 2
2. else if Type(`settings`) is Object
   a. If Type(`settings.ignoreComments`) is Boolean,
i. Let XML.ignoreComments = settings.ignoreComments
b. If Type(settings.ignoreProcessingInstructions) is Boolean
   i. Let XML.ignoreProcessingInstructions = settings.ignoreProcessingInstructions
c. If Type(settings.ignoreWhitespace) is Boolean
   i. Let XML.ignoreWhitespace = settings.ignoreWhitespace
d. If Type(settings.prettyPrinting) is Boolean
   i. Let XML.prettyPrinting = settings.prettyPrinting
e. If Type(settings.prettyIndent) is Number
   i. Let XML.prettyIndent = settings.prettyIndent

3. Return

12.4.3.9 XML.defaultSettings()

The defaultSettings method is a convenience method for managing the collection of global XML settings stored as properties of the XML constructor (sections 12.4.3.2 through 12.4.3.6). It may be used to obtain an object containing the default XML settings. This object may be inspected to determine the default settings or be passed as an argument to the setSettings method to restore the default XML settings.

Semantics

When the defaultSettings method of the XML constructor is called with no arguments, the following steps are taken:

1. Let s be a new Object created as if by calling the constructor new Object()
2. Let s.ignoreComments = true
3. Let s.ignoreProcessingInstructions = true
4. Let s.ignoreWhitespace = true
5. Let s.prettyPrinting = true
6. Let s.prettyIndent = 2
7. Return s

12.4.4 Properties of the XML Prototype Object (Built-in Methods)

Each value of type XML has a set of built-in methods available for performing common operations. These built-in methods are properties of the XML prototype object and are described in the following sections.

The XML prototype object is itself an XML object (its [[Class]] property is "text") whose value is the empty string.

The value of the internal [[Prototype]] property of the XML prototype object is the Object prototype object. (section 15.2.3.1 of ECMAScript Edition 3).

12.4.4.1 XML.prototype.constructor

The initial value of the XML.prototype.constructor is the built-in XML constructor.

12.4.4.2 XML.prototype.addNamespace( namespace )

Overview

The addNamespace method adds a namespace declaration to the in scope namespaces for this XML object and returns this XML object. If the in scope namespaces for the XML object already contains a namespace with a prefix matching that of the given parameter, the prefix of the existing namespace is set to undefined.

Semantics

When the setNamespace method of an XML object x is called with one parameter namespace, the following step is taken:

1. Let ns a Namespace constructed as if by calling the function Namespace(namespace)
2. Call the [[AddInScopeNamespace]] method of x with parameter ns
3. Return x
12.4.4.3 XML.prototype.appendChild ( child )

Overview

The appendChild method appends a deep copy of the given child to the end of this XML object’s properties and returns this XML object. For example,

```javascript
var e = <employees>
    <employee id="0">Jim</employee><age>25</age>
    <employee id="1">Joe</employee><age>20</age>
</employees>;

// Add a new child element to the end of Jim's employee element
e.employee.(name == "Jim").appendChild(<hobby>snorkeling</hobby>);
```

Semantics

When the appendChild method of an XML object x is called with one parameter child, the following steps are taken:

1. Call the [[Put]] method of x with arguments x.[[Length]] and child
2. Return x

12.4.4.4 XML.prototype.attribute ( attributeName )

Overview

The attribute method returns an XMLList containing zero or one XML attributes associated with this XML object that have the given attributeName. For example,

```javascript
// get the id of the employee named Jim
e.employee.(name == "Jim").attribute("id");
```

Semantics

When the attribute method of an XML object x is called with a parameter attributeName, the following steps are taken:

1. Let name = ToAttributeName(attributeName)
2. Return the result of calling the [[Get]] method of x with argument name

12.4.4.5 XML.prototype.attributes ( )

Overview

The attributes method returns an XMLList containing the XML attributes of this object. For example,

```javascript
// print the attributes of an XML object
function printAttributes(x) {
    for (var a in x.attributes()) {
        print("The attribute named " + a.name() + " has the value " + a);
    }
}
```

Semantics

When the attributes method of an XML object x is called, the following step is taken:

1. Return the result of calling the [[Get]] method of x with argument ToAttributeName("*")
12.4.4.6 XML.prototype.child (propertyName)

Overview

The child method returns the list of children in this XML object matching the given `propertyName` results.

Semantics

When the child method of an XML object \( x \) is called, it performs the following step:

1. Return the result of calling the \([\text{Get}]\) method of \( x \) with argument `propertyName`

12.4.4.7 XML.prototype.childIndex ( )

Overview

The childIndex method returns a Number representing the ordinal position of this XML object within the context of its parent. For example,

```
// Get the ordinal index of the employee named Joe.
var joeindex = e.employee.(name == "Joe").childIndex();
```

Semantics

When the childIndex method of an XML object \( x \) is called, it performs the following steps:

1. Let \( parent = x.[\text{Parent}] \)
2. If \( (parent == \text{null}) \) or \( (x.[\text{Class}] == \"attribute\") \), return \( \text{NaN} \)
3. Let \( q \) be the property of \( parent \), where \( parent[q] \) is the same object as \( x \)
4. Return \( \text{ToNumber}(q) \)

12.4.4.8 XML.prototype.children ( )

Overview

The children method returns an XMLList containing all the properties of this XML object in order. For example,

```
// Get child elements of first employee: returns an XMLList containing:
// <name>Jim</name>, <age>25</age> and <hobby>Snorkeling</hobby>
var emps = e.employee[0].children();
```

Semantics

When the children method of an XML object \( x \) is called, it performs the following step:

1. Return the results of calling the \([\text{Get}]\) method of \( x \) with argument `*`

12.4.4.9 XML.prototype.comment ( )

Overview

The comment method returns an XMLList containing the properties of this XML object that represent XML comments.

Semantics

When the comment method of an XML object \( x \) is called, it performs the following steps:

1. Let \( l \) be a new XMLList with \( l.[\text{TargetObject}] = x \) and \( l.[\text{TargetProperty}] = \text{null} \)
2. For \( i = 0 \) to \( x.[\text{Length}]-1 \)
   a. If \( x[i].[\text{Class}] == \"comment\" \), call the \([\text{Append}]\) method of \( l \) with argument \( x[i] \)
3. Return `l`

12.4.4.10 `XML.prototype.copy()`

**Overview**

The `copy` method returns a deep copy of this XML object with the internal `[[Parent]]` property set to `null`.

**Semantics**

When the `copy` method is called on an XML object `x`, the following steps are taken:

1. Return the result of calling the `[[DeepCopy]]` method of `x`

12.4.4.11 `XML.prototype.descendants([name])`

**Overview**

The `descendants` method returns all the XML valued descendants (children, grandchildren, great-grandchildren, etc.) of this XML object with the given `name`. If the `name` parameter is omitted, it returns all descendants of this XML object.

**Semantics**

When the `descendants` method is called on an XML object `x` with the optional parameter `name`, the following steps are taken:

1. If `name` is not specified, let `name = "*"`
2. Return the result of calling the `[[Descendants]]` method of `x` with argument `name`

12.4.4.12 `XML.prototype.inScopeNamespaces()`

**Overview**

The `inScopeNamespaces` method returns an Array of Namespace objects representing the namespaces in scope for this XML object in the context of its parent. If the parent of this XML object is modified, the associated namespace declarations may change.

**Semantics**

When the `inScopeNamespaces` method is called on an XML object `x`, the following steps are taken:

1. Let `y = x`
2. Let `inScopeNS = { }`
3. While (`y` is not `null`)
   a. For each `ns in y.[[InScopeNamespaces]]`
      i. If there exists no namespace `n ∈ inScopeNS`, such that `n.prefix == ns.prefix`
   1. Let `inScopeNS = inScopeNS ∪ ns`
   b. Let `y = y.[[Parent]]`
4. Note: The E4X data model does not enforce the constraint: ∀ `x ∈ XML : x.[[InScopeNamespaces]] ⊇ x.[[Parent]].[[InScopeNamespaces]]`. However, implementations that do enforce this constraint may set `inScopeNS = x.[[InScopeNamespaces]]` instead of using the computation above.
5. Let `a` be a new Array created as if by calling the constructor, new Array()
6. Let `i = 0`
7. For each `ns ∈ inScopeNS`
   a. Call the `[[Put]]` method of `a` with arguments `ToString(i)` and `ns`
   b. Let `i = i + 1`
8. Return `a`

12.4.4.13 `XML.prototype.insertChildAfter(child1, child2)`

**Overview**


The insertChildAfter method inserts the given child2 after the given child1 in this XML object and returns this XML object. If child1 is null, the insertChildAfter method inserts child2 before all children of this XML object (i.e., after none of them). If child1 does not exist in this XML object, it throws a TypeError exception.

Semantics

When the insertChildAfter method is called on an XML object x with parameters child1 and child2, the following steps are taken:

1. If x.[[Class]] ∈ {"text", "comment", "processing-instruction", "attribute"}, throw a TypeError exception
2. If (child1 == null)
   a. Call the [[Insert]] method of x with arguments "0" and child2
   b. Return x
3. Else if Type(child1) is XML
   a. For i = 0 to x.[[Length]]-1
      i. If x[i] is the same object as child1
         1. Call the [[insert]] method of x with arguments ToString(i + 1) and child2
         2. Return x
4. Throw a TypeError exception

12.4.4.14 XML.prototype.insertChildBefore (child1, child2)

Overview

The insertChildBefore method inserts the given child2 before the given child1 in this XML object and returns this XML object. If child1 is null, the insertChildBefore method inserts child2 after all children in this XML object (i.e., before none of them). If child1 does not exist in this XML object, it throws a TypeError exception.

Semantics

When the insertChildBefore method is called on an XML object x with parameters child1 and child2, the following steps are taken:

1. If x.[[Class]] ∈ {"text", "comment", "processing-instruction", "attribute"}, throw a TypeError exception
2. If (child1 == null)
   a. Call the [[Insert]] method of x with arguments ToString(x.[[Length]]) and child2
   b. Return x
3. Else if Type(child1) is XML
   a. For i = 0 to x.[[Length]]-1
      i. If x[i] is the same object as child1
         1. Call the [[insert]] method of x with arguments ToString(i) and child2
         2. Return x
4. Throw a TypeError exception

12.4.4.15 XML.prototype.hasOwnProperty (P)

Overview

The hasOwnProperty method returns a Boolean value indicating whether this object has the property specified by P. For all XML objects except the XML prototype object, this is the same result returned by the internal method [[HasProperty]]. For the XML prototype object, hasOwnProperty also examines the list of local properties to determine if there is a method property with the given name.

Semantics

When the hasOwnProperty method of an XML object x is called with parameter P, the following step is taken:

1. If the result of calling the [[HasProperty]] method of this object with argument P is true, return true
2. If x has a property with name ToString(P), return true
3. Return false
12.4.4.16 __XML.prototype.hasComplexContent( )__

**Overview**

The hasComplexContent method returns a Boolean value indicating whether this XML object contains complex content. An XML object is considered to contain complex content if it represents an XML element that has child elements. XML objects representing attributes, comments, processing instructions and text nodes do not have complex content. The existence of attributes, comments, processing instructions and text nodes within an XML object is not significant in determining if it has complex content.

**Semantics**

When the hasComplexContent method is called on an XML object `x`, the following steps are taken:

1. If `x.[[Class]]` ∈ {"attribute", "comment", "processing-instruction", "text"}, return false
2. For each property `p` in `x`
   a. If `x[p].[[Class]]` === "element", return true
3. Return false

12.4.4.17 __XML.prototype.hasSimpleContent( )__

**Overview**

The hasSimpleContent method returns a Boolean value indicating whether this XML object contains simple content. An XML object is considered to contain simple content if it represents a text node, represents an attribute node or if it represents an XML element that has no child elements. XML objects representing comments and processing instructions do not have simple content. The existence of attributes, comments, processing instructions and text nodes within an XML object is not significant in determining if it has simple content.

**Semantics**

When the hasSimpleContent method is called on an XML object `x`, the following steps are taken:

1. If `x.[[Class]]` ∈ {"comment", "processing-instruction"}, return false
2. For each property `p` in `x`
   a. If `x[p].[[Class]]` === "element", return false
3. Return true

12.4.4.18 __XML.prototype.length ( )__

**Overview**

The length method returns the number of properties in this XML object. For example,

```javascript
// print each child element of the first employee element stored in e
for (var i = 0; i < e.employee[0].length(); i++) {
    print("Child element:" + e.employee[0][i]);
}
```

**Semantics**

When the length method is called on an XML object `x`, the following step is taken:

1. Return `x.[[Length]]`

12.4.4.19 __XML.prototype.localName ( )__

**Overview**

The localName method returns the local name portion of the qualified name of this XML object.

**Semantics**

When the localName method is called on an XML object `x`, the following step is taken:

1. Return `x.[[LocalName]]`
When the localName method is called on an XML object \( x \), the following step is taken:

1. If \( x.\text{[[Name]]} == \text{null} \), return \( \text{null} \)
2. Return \( x.\text{[[Name]].localName} \)

12.4.4.20  XML.prototype.name( )

Overview

The name method returns the qualified name associated with this XML object.

Semantics

When the name method is called on an XML object \( x \), the following step is taken:

1. Return \( x.\text{[[Name]]} \)

12.4.4.21  XML.prototype.namespace([prefix])

Overview

If no prefix is specified, the namespace method returns the Namespace associated with the qualified name of this XML object.

If a prefix is specified, the namespace method looks for a namespace in scope for this XML object with the given prefix and, if found, returns it. If no such namespace is found, the namespace method returns \text{undefined}.

Semantics

When the namespace method is called on an XML object \( x \) with zero arguments or one argument \( \text{prefix} \), the following steps are taken:

2. Let \( y = x \)
3. Let \( \text{inScopeNS} = \{ \} \)
4. While \( y \) is not \( \text{null} \)
   a. For each \( n \) in \( y.\text{[[InScopeNamespaces]]} \)
      i. If there exists no namespace \( n \in \text{inScopeNS} \), such that \( n.\text{prefix} == \text{ns.prefix} \)
         1. Let \( \text{inScopeNS} = \text{inScopeNS} \cup n \)
   b. Let \( y = y.\text{[[Parent]]} \)
5. Note: The E4X data model does not enforce the constraint: \( \forall x \in \text{XML} : x.\text{[[InScopeNamespaces]]} \supseteq x.\text{[[Parent]].[[InScopeNamespaces]]} \). However, implementations that do enforce this constraint may set \( \text{inScopeNS} = x.\text{[[InScopeNamespaces]]} \) instead of using the computation above.
6. If \( \text{prefix} \) was not specified
   a. If \( x.\text{[[Class]]} \in \{ \text{"text"}, \text{"comment"}, \text{"processing-instruction"} \} \), return \( \text{null} \)
   b. Return the result of calling the \( [[\text{GetNamespace}]] \) method of \( x.\text{[[Name]]} \) with argument \( \text{inScopeNS} \)
7. Else
   a. Let \( \text{prefix} = \text{ToString}(\text{prefix}) \)
   b. Find a Namespace \( ns \in \text{inScopeNS} \), such that \( ns.\text{prefix} = \text{prefix} \). If no such \( ns \) exists, let \( ns = \text{undefined} \).
   c. Return \( ns \)

12.4.4.22  XML.prototype.namespaceDeclarations( )

Overview

The namespaceDeclarations method returns an Array of Namespace objects representing the namespace declarations associated with this XML object in the context of its parent. If the parent of this XML object is modified, the associated namespace declarations may change.

Semantics
When the namespaceDeclarations method is called on an XML object `x`, the following steps are taken:

1. Let `a` be a new Array created as if by calling the constructor, `new Array()`
2. If `x.[[Class]] ∈ {"text", "comment", "processing-instruction", "attribute"}`, return `a`
3. Let `y = x.[[Parent]]`
4. Let `ancestorNS = {}`
5. While (`y` is not `null`)
   a. For each `ns` in `y.[[InScopeNamespaces]]`
      i. If there exists no namespace `n ∈ ancestorNS`, such that `n.prefix == ns.prefix`
         1. Let `ancestorNS = ancestorNS ∪ ns`
   b. Let `y = y.[[Parent]]`
6. Note: The E4X data model does not enforce the constraint: \( ∀ x ∈ XML : x.[[InScopeNamespaces]] ⊇ x.[[Parent]].[[InScopeNamespaces]] \). However, implementations that do enforce this constraint may set `ancestorNS = x.[[Parent]].[[InScopeNamespaces]]` instead of using the computation above.
7. Let `declaredNS = { }`
8. for each `ns` in `x.[[InScopeNamespaces]]`
   a. if there exists no namespace `n ∈ ancestorNS`, such that `n.prefix == ns.prefix` and `n.uri == ns.uri`
      i. Let `declaredNS = declaredNS ∪ ns`
9. Let `i = 0`
10. For each `ns ∈ declaredNS`
    a. Call the `[[Put]]` method of `a` with arguments `ToString(i)` and `ns`
    b. Let `i = i + 1`
11. Return `a`

### 12.4.4.23 XML.prototype.nodeKind ( )

**Overview**

The nodeKind method returns a string representing the `[[Class]]` of this XML object.

**Semantics**

When the nodeKind method is called on an XML object `x`, the following step is taken:

1. Return `x.[[Class]]`

### 12.4.4.24 XML.prototype.normalize ( )

**Overview**

The normalize method puts all text nodes in this and all descendant XML objects into a normal form by merging adjacent text nodes and eliminating empty text nodes. It returns this XML object.

**Semantics**

When the normalize method is called on an XML object `x`, the following steps are taken:

1. Let `i = 0`
2. While `i < x.[[Length]]`
   a. If `x[i].[[Class]] == "element"
      i. Call the normalize method of `x[i]`
      ii. Let `i = i + 1`
   b. Else if `x[i].[[Class]] == "text"
      i. While `(i+1) < x.[[Length]]` and `x[i+1].[[Class]] == "text"`
         1. Let `x[i].[[Value]]` be the result of concatenating `x[i].[[Value]]` and `x[i+1].[[Value]]`
         2. Call the `[[Delete]]` method of `x` with argument `ToString(i + 1)`
      ii. If `x[i].[[Value]].length == 0`
          1. Call the `[[Delete]]` method of `x` with argument `ToString(i)`
      iii. Else
          1. Let `i = i + 1`
c. Else
   i. Let \( i = i + 1 \)
3. Return \( x \)

12.4.4.25 \( \text{XML.prototype.parent}() \)

Overview

The parent method returns the parent of this XML object. For example,

```javascript
// Get the parent element of the second name in "e". Returns <employee id="1" ... var firstNameParent = e.name[1].parent()
```

Semantics

When the parent method is called on an XML object \( x \), the following step is taken:

1. Return \( x.\{\text{Parent}\} \)

12.4.4.26 \( \text{XML.prototype.processingInstruction}([\text{name}]\) )

Overview

When the processingInstruction method is called with one parameter \( \text{name} \), it returns an XMLList containing all the children of this XML object that are processing-instructions with the given \( \text{name} \). When the processingInstruction method is called with no parameters, it returns an XMLList containing all the children of this XML object that are processing-instructions regardless of their name.

Semantics

When the processingInstruction method is called on an XML object \( x \) with optional parameter \( \text{name} \), the following steps are taken:

1. If \( \text{name} \) is not specified, let \( \text{name} = "\*" \)
2. Let \( \text{name} = \text{ToXMLName}(\text{name}) \)
3. Let \( l = \) a new XMLList with \( l.\{\text{TargetObject}\} = x \) and \( l.\{\text{TargetProperty}\} = \text{null} \)
4. For \( i = 0 \) to \( x.\{\text{Length}\} - 1 \)
   a. If \( x[i].\{\text{Class}\} == "\text{processing-instruction}" \)
      i. If \( \text{name}.\text{localName} == "\*" \) or \( \text{name}.\text{localName} == x[i].\{\text{Name}\}.\text{localName} \)
         1. Call the \([\text{Append}]\) method of \( l \) with argument \( x[i] \)
5. Return \( l \)

12.4.4.27 \( \text{XML.prototype.prependChild} (\text{value})\)

Overview

The prependChild method inserts a deep copy of the given \( \text{child} \) into this object prior to its existing XML properties and returns this XML object. For example,

```javascript
// Add a new child element to the front of John's employee element
e.employee.(name == "John").prependChild(<prefix>Mr.</prefix>);
```

Semantics

When the prependChild method is called on an XML object \( x \) with parameter \( \text{value} \), the following steps are taken:

1. Call the \([\text{Insert}]\) method of this object with arguments "0" and \( \text{value} \)
2. Return \( x \)
12.4.4.28 **XML.prototype.propertyIsEnumerable ( P )**

**Overview**

The propertyIsEnumerable method returns a Boolean value indicating whether the property P is accessible via the XML [[Get]] method. All properties of XML objects accessible via the XML [[Get]] method are enumerable.

**Semantics**

When the propertyIsEnumerable method of an XML object x is called with parameter P, the following step is taken:

1. Return the result of calling the [[HasProperty]] method of this object with argument P.

12.4.4.29 **XML.prototype.removeNamespace ( namespace )**

**Overview**

The removeNamespace method removes the given namespace from the in scope namespaces of this object and all its descendents, then returns a copy of this XML object. The removeNamespaces method will not remove a namespace from an object where it is referenced by that object’s QName or the QNames of that object’s attributes.

**Semantics**

When the removeNamespace method is called on an XML object x with parameter namespace, the following steps are taken:

2. If x.[[Class]] ∈ {"text", "comment", "processing-instruction", "attribute"}, return x
3. Let ns be a Namespace object created as if by calling the function Namespace( namespace )
4. Let thisNS be the result of calling [[GetNamespace]] on x.[[Name]] with argument x.[[InScopeNamespaces]]
5. If (thisNS == ns), return x
6. For each attribute a ∈ x.[[Attributes]]
   a. Let aNS be the result of calling [[GetNamespace]] on a.[[Name]] with argument x.[[InScopeNamespaces]]
   b. If (aNS == ns), return x
7. If ns.prefix == undefined
   a. If there exists a namespace n ∈ x.[[InScopeNamespaces]], such that n.uri == ns.uri, remove the namespace n from x.[[InScopeNamespaces]]
   b. Else
      a. If there exists a namespace n ∈ x.[[InScopeNamespaces]], such that n.uri == ns.uri and n.prefix == ns.prefix, remove the namespace n from x.[[InScopeNamespaces]]
5. For each property p of x
   a. If p.[[Class]] = "element", call the removeNamespace method of p with argument ns
10. Note: The E4X data model does not enforce the constraint: ∀ x ∈ XML : x.[[InScopeNamespaces]] ⊇ x.[[Parent]].[[InScopeNamespaces]]. However, implementations may at this point remove namespace from the [[InScopeNamespaces]] property of any ancestors of x. provided namespace is not referenced by the associated ancestor's QName or the QNames of the associated ancestor's attributes.
11. Return x

12.4.4.30 **XML.prototype.replace ( propertyName , value )**

**Overview**

The replace method replaces the XML properties of this XML object specified by propertyName with value and returns this XML object. Unlike the assignment operator, the replace method does not give special treatment to cases where value contains simple content. It always replaces the specified properties with the given value. If this XML object contains no properties that match propertyName, the replace method returns without modifying this XML object. The propertyName parameter may be a numeric property name, an unqualified name for a set of XML elements, a qualified name for a set of XML elements or the properties wildcard "*". When the propertyName parameter is an unqualified name, it identifies XML elements in the default namespace. The value parameter may be an XML object, XMLList object or any value that may be converted to a String with ToString(). For example,
// Replace the first employee record with an open staff requisition
employees.replace(0, <requisition status="open"/>)

// Replace all item elements in the order with a single empty item
order.replace("item", <item/>)

Semantics

When the replace method is called on an XML object \( x \) with parameters \( propertyName \) and \( value \), the following steps are taken:

1. If \( x.\left[ \text{Class} \right] \in \{ \text{"text"}, \text{"comment"}, \text{"processing-instruction"}, \text{"attribute"} \} \), return \( x \)
2. If \( \text{Type}(\text{value}) \notin \{ \text{XML}, \text{XMLList} \} \), let \( c = \text{ToString}(\text{value}) \)
3. Else let \( c \) be the result of calling the [[DeepCopy]] method of \( \text{value} \)
4. If \( \text{ToString}(\text{ToUint32}(\text{P})) = \text{P} \)
   a. Call the [[Replace]] method of \( x \) with arguments \( \text{P} \) and \( c \) and return \( x \)
5. Let \( n \) be a QName object created as if by calling the function QName(\( \text{P} \))
6. Let \( i \equiv \text{undefined} \)
7. For \( k = 0 \) to \( x.\left[ \text{Length} \right]-1 \)
   a. If \( ((n.\text{localName} = \text{"*"})
      \text{or}( (x[k].\left[ \text{Class} \right] = \text{"element"}) \text{and}( x[k].\left[ \text{Name} \right].\text{localName} = n.\text{localName} ))
      \text{and}( (n.\text{uri} = \text{undefined}) \text{or}( n.\text{uri} \equiv x[k].\left[ \text{Name} \right].\text{uri} ))
     
     i. If \( i = \text{undefined} \), let \( i = k \)
     ii. Else call the [[Delete]] method of \( x \) with argument \( \text{ToString}(k) \)
   b. If \( i = \text{undefined} \), return \( x \)
   c. Call the [[Replace]] method of \( x \) with arguments \( \text{ToString}(i) \) and \( c \)
8. If \( i = \text{undefined} \), return \( x \)
9. Call the [[Replace]] method of \( x \) with arguments \( \text{ToString}(i) \) and \( c \)
10. Return \( x \)

12.4.4.31 XML.prototype.setInnerXML ( value )

Overview

The setInnerXML method replaces the XML properties of this XML object with a new set of XML properties from \( \text{value} \). \( \text{value} \) may be a single XML object or an XMLList. setInnerXML returns this XML object. For example,

// Replace the entire contents of Jim's employee element
e.employee.(name == "Jim").setInnerXML(<name>John</name> + <age>35</age>);

Semantics

When the setInnerXML method is called on an XML object \( x \) with parameter \( \text{value} \), the following steps are taken:

1. Call the [[Put]] method of \( x \) with arguments "*" and \( \text{value} \)
2. Return \( x \)

12.4.4.32 XML.prototype.setLocalName ( name )

Overview

The setLocalName method replaces the local name this XML object with a string constructed from the given \( \text{name} \).

Semantics

When the setLocalName method is called on an XML object \( x \) with parameter \( \text{name} \), the following steps are taken:

1. If \( x.\left[ \text{Class} \right] \in \{ \text{"text"}, \text{"comment"} \} \), return
2. if \( \text{Type}(\text{name}) \) is Object and \( (\text{name}.\left[ \text{Class} \right] = \text{"QName"}) \)
   a. Let \( \text{name} = \text{name}.\text{localName} \)
3. else
a. Let name = ToString(name)
4. Let x.[[Name]].localName = name

12.4.4.33  XML.prototype.setName ( name )

Overview

The setName method replaces the name this XML object with a QName or AttributeName constructed from the given name.

Semantics

When the setName method is called on an XML object x with parameter name, the following steps are taken:

1. If x.[[Class]] ∈ {"text", "comment"}, return
2. if (Type(name) is Object) and (name.[[Class]] == "QName") and (name.uri == undefined)
   a. Let name = name.localName
3. Let n be a new QName created if by calling the constructor new QName(name)
4. if x.[[Class]] == "processing-instruction", let n.uri be the empty string
5. Let x.[[Name]] = n

12.4.4.34  XML.prototype.setNamespace ( ns )

Overview

The setNamespace method replaces the namespace associated with the name of this XML object with the given namespace.

Semantics

When the setNamespace method is called on an XML object x with parameter ns, the following step is taken:

1. If x.[[Class]] ∈ {"text", "comment", "processing-instruction"}, return
2. Let namespace be a new Namespace created as if by calling the constructor new Namespace(ns)
3. Call [[SetNamespace]] on x.[[Name]] with argument namespace

12.4.4.35  XML.prototype.text ( )

Overview

The text method returns an XMLList containing all XML properties of this XML object that represent XML text nodes.

Semantics

When the text method of an XML object x is called, the following steps are taken:

1. Let l be a new XMLList with l.[[TargetObject]] = x and l.[[TargetProperty]] = null
2. For i = 0 to x.[[Length]]-1
   a. If x[i].[[Class]] == "text", Call the [[Append]] method of l with argument x[i]
3. Return l

12.4.4.36  XML.prototype.toString ( )

Overview

The toString method returns a string representation of this XML object per the ToString conversion operator described in section 9.1.

Semantics

When the toString method of an XML object x is called, the following step is taken:

1. Return ToString(x)
The toXMLString() method returns an XML encoded string representation of this XML object per the ToXMLString conversion operator described in section 9.2. Unlike the toString method, toXMLString provides no special treatment for XML objects that contain only XML text nodes (i.e., primitive values). The toXMLString method always includes the start tag, attributes and end tag of the XML object regardless of its content. It is provided for cases when the default XML to string conversion rules are not desired. For example,

```javascript
var item = <item>
  <description>Laptop Computer</description>
  <price>2799.95</price>
  <quantity>1</quantity>
</item>;
// returns "Description stored as <description>Laptop Computer</description"
var logmsg = "Description stored as " + item.description.toXMLString();
// returns "Thank you for purchasing a Laptop Computer!" (with tags removed)
var message = "Thank you for purchasing a " + item.description + "!";
```

**Semantics**

When the toXMLString method of an XML object \(x\) is called, the following step is taken:

1. Return ToXMLString(\(x\))

### 12.4.4.38 XML.prototype.valueOf ( )

**Overview**

The valueOf method returns this XML object.

**Semantics**

When the valueOf method of an XML object \(x\) is called, the following step is taken:

1. Return \(x\)

### 12.4.5 Properties of XML Instances

XML instances have no special properties beyond those inherited from the XML prototype object.

### 12.5 XMLList Objects

#### 12.5.1 The XMLList Constructor Called as a Function

**Syntax**

```javascript
XMLList ( value )
```

**Overview**

When XMLList is called as a function rather than as a constructor, it converts its argument to an XMLList object. When its argument is an XMLList, the input argument is returned without modification.

**Semantics**
When XMLList is called as a function with parameter `value`, the following steps are taken:

1. If `value` is `null`, `undefined` or not supplied, let `value` be the empty string
2. Return `ToXMLList(value)`

### 12.5.2 The XMLList Constructor

**Syntax**

```
new XMLList([value])
```

**Overview**

When XMLList is called as part of a new expression, it is a constructor and creates a new XMLList object. When the XMLList constructor is called with no arguments, it returns an empty XMLList. When the XMLList constructor is called with a `value` of type XMLList, the XMLList constructor returns a shallow copy of the `value`. When the XMLList constructor is called with a non-XMLList value, it converts its input argument to an XMLList object.

**Semantics**

When the XMLList constructor is called with an optional parameter `value`, the following steps are taken:

1. If `value` is `null`, `undefined` or not supplied, let `value` be the empty string
2. If `Type(value)` is XMLList
   a. Let `l` be a new XMLList object with `l.[[TargetObject]] = null`
   b. Call the `[[Append]]` method of `l` with argument `value`
   c. Return `l`
3. Else
   a. Return `ToXMLList(value)`

### 12.5.3 Properties of the XMLList Constructor

The value of the internal `[[Prototype]]` property of the XMLList constructor is the Function prototype object.

Besides the internal properties and the length property (whose value is 1), the XMLList constructor has the following properties:

#### 12.5.3.1 XMLList.prototype

The initial value of the XMLList.prototype property is the XMLList prototype object (section 12.5.4).

This property has the attributes { DontEnum, DontDelete, ReadOnly }.

#### 12.5.4 Properties of the XMLList Prototype Object (Built-in Methods)

Each value of type XMLList has a set of built-in methods available for performing common operations. These built-in methods are described in the following sections.

The XMLList prototype object is itself an XMLList object (its `[[Class]]` property is "XMLList") whose value is the empty XMLList.

The value of the internal `[[Prototype]]` property of the XMLList prototype object is the Object prototype object. (section 15.2.3.1 of ECMA-Script Edition 3).

Note: To simplify the programmer’s task, E4X intentionally blurs the distinction between a single XML object and an XMLList containing only one value. To this end, E4X extends the ECMAScript function call semantics such that all methods available for values of type XML are also available for XMLLists of size one. See section 10.2.2 for more information.
12.5.4.1 XMLList.prototype.constructor
The initial value of the XMLList prototype constructor is the built-in XMLList constructor.

12.5.4.2 XMLList.prototype.attribute ( attributeName )
Overview
The attribute method calls the attribute method of each XML object in this XMLList object passing attributeName as a parameter and returns an XMLList containing the results in order.

Semantics
When the attribute method is called on an XMLList object l with parameter attributeName, the following steps are taken:

1. Let name = ToAttributeName(attributeName)
2. Return the result of calling the [[Get]] method of l with argument name

12.5.4.3 XMLList.prototype.attributes ( )
Overview
The attributes method calls the attributes() method of each XML object in this XMLList object and returns an XMLList containing the results in order.

Semantics
When the attributes method is called on an XMLList object l, the following step is taken:

1. Return the result of calling the [[Get]] method of l with argument ToAttributeName("*")

12.5.4.4 XMLList.prototype.child ( propertyName )
Overview
The child method calls the child() method of each XML object in this XMLList object and returns an XMLList containing the results in order.

Semantics
When the child method is called on an XMLList object l with parameter propertyName, the following step is taken:

1. Return the result of calling the [[Get]] method of l with argument propertyName

12.5.4.5 XMLList.prototype.children ( )
Overview
The children method calls the children() method of each XML object in this XMLList object and returns an XMLList containing the results concatenated in order. For example,

```javascript
// get all the children of all the items in the order
var allItemChildren = order.item.children();

// get all grandchildren of the order that have the name price
var grandChildren = order.children().price;
```

Semantics
When the children method is called on an XMLList object l, the following step is taken:
1. Return the results of calling the [[Get]] method of \( l \) with argument "*"

### 12.5.4.6 XMLList.prototype.comment ( )

#### Overview

The comment method calls the comment method of each XML object in this XMLList object and returns an XMLList containing the results concatenated in order.

#### Semantics

When the comment method is called on an XMLList object \( l \), the following steps are taken:

1. Let \( m \) be a new XMLList with \( m.[[TargetObject]] = l \)
2. For \( i = 0 \) to \( l.[[Length]]-1 \)
   a. If \( l[i].[[Class]] === "element" \)
      i. Let \( r = l[i].comment() \)
      ii. If \( r.[[Length]] > 0 \), call the [[Append]] method of \( m \) with argument \( r \)
3. Return \( m \)

### 12.5.4.7 XMLList.prototype.copy ( )

#### Overview

The copy method returns a deep copy of this XMLList object.

#### Semantics

When the copy method is called on an XMLList object \( l \), the following step is taken:

1. Return the result of calling the [[DeepCopy]] method of \( l \)

### 12.5.4.8 XMLList.prototype.descendants ( [ name ] )

The descendants method calls the descendants method of each XML object in this XMLList object with the optional parameter \( name \) (or the string "*" if \( name \) is omitted) and returns an XMLList containing the results concatenated in order.

#### Semantics

When the descendants method is called on an XMLList object \( l \) with optional parameter \( name \), the following steps are taken:

1. If \( name \) is not specified, \( name = "*" \)
2. Return the result of calling the [[Descendants]] method of \( l \) with argument \( name \)

### 12.5.4.9 XMLList.prototype.hasOwnProperty ( P )

#### Overview

The hasOwnProperty method returns a Boolean value indicating whether this object has the property specified by \( P \). For all XMLList objects except the XMLList prototype object, this is the same result returned by the internal method[[HasProperty]]. For the XMLList prototype object, hasOwnProperty also examines the list of local properties to determine if there is a method property with the given name.

#### Semantics

When the hasOwnProperty method of an XMLList object \( x \) is called with parameter \( P \), the following step is taken:

1. If the result of calling the [[HasProperty]] method of this object with argument \( P \) is \( true \), return \( true \)
2. If \( x \) has a property with name \( Tostring(P) \), return \( true \)
3. Return \( false \)
12.5.4.10  XMLList.prototype.hasComplexContent( )

Overview

The hasComplexContent method returns a Boolean value indicating whether this XMLList object contains complex content. An XMLList object is considered to contain complex content if it is not empty, contains a single XML item with complex content or contains elements.

Semantics

When the hasComplexContent method is called on an XMLList object x, the following steps are taken:

1. If x.[[Length]] == 0, return false
2. If x.[[Length]] == 1, return x[0].hasComplexContent()
3. For each property p in x
   a. If x[p].[[Class]] == "element", return true
4. Return false

12.5.4.11  XMLList.prototype.hasSimpleContent( )

Overview

The hasSimpleContent method returns a Boolean value indicating whether this XMLList contains simple content. An XMLList object is considered to contain simple content if it is empty, contains a single XML item with simple content or contains no elements.

Semantics

When the hasSimpleContent method is called on an XMLList object x, the following steps are taken:

1. If x.[[Length]] == 0, return true
2. If x.[[Length]] == 1, return x[0].hasSimpleContent()
3. For each property p in x
   a. If x[p].[[Class]] == "element", return false
4. Return true

12.5.4.12  XMLList.prototype.length ( )

Overview

The length method returns the number of properties in this XMLList object. For example,

for (var i = 0; i < e..name.length(); i++) {
  print("Employee name:" + e..name[i]);
}

Semantics

When the length method of an XMLList object l is called, the following step is taken:

1. Return l.[[Length]]

12.5.4.13  XMLList.prototype.normalize ( )

Overview

The normalize method puts all text nodes in this XMLList, all the XML objects it contains and the descendents of all the XML objects it contains into a normal form by merging adjacent text nodes and eliminating empty text nodes. It returns this XMLList object.

Semantics
When the normalize method is called on an XMLList object \( l \), the following steps are taken:

1. Let \( i = 0 \)
2. While \( i < l.[[\text{Length}]] \)
   a. If \( l[i].[[\text{Class}]] == "\text{element}" \)
      i. Call the normalize method of \( l[i] \)
      ii. Let \( i = i + 1 \)
   b. Else if \( l[i].[[\text{Class}]] == "\text{text}" \)
      i. While \( ((i+1) < l.[[\text{Length}]] \) and \( l[i+1].[[\text{Class}]] == "\text{text}" \)
         1. Let \( l[i].[[\text{Value}]] \) be the result of concatenating \( l[i].[[\text{Value}]] \) and \( l[i+1].[[\text{Value}]] \)
         2. Call the [[Delete]] method of \( l \) with argument ToString\( (i+1) \)
      ii. If \( l[i].[[\text{Value}]].\text{length} == 0 \)
          1. Call the [[Delete]] method of \( l \) with argument ToString\( (i) \)
      iii. Else
          1. Let \( i = i + 1 \)
   c. Else
      i. Let \( i = i + 1 \)
3. Return \( l \)

12.5.4.14  XMLList.prototype.parent ( )

**Overview**

If all items in this XMLList object have the same parent, it is returned. Otherwise, the parent method returns \text{undefined}.

**Semantics**

When the parent method is called on an XMLList object \( l \), the following steps are taken:

1. If \( l.[[\text{Length}]] = 0 \), return \text{undefined}
2. Let \( \text{parent} = l[0].[[\text{Parent}]] \)
3. For \( i = 1 \) to \( l.[[\text{Length}]]-1 \), if \( l[i].[[\text{Parent}]] \) is not equal to \( \text{parent} \), return \text{undefined}
4. Return \( \text{parent} \)

12.5.4.15  XMLList.prototype.processingInstruction ( [ name ] )

**Overview**

The processingInstruction method calls the processingInstruction method of each XML object in this XMLList object passing the optional parameter \text{name} (or "" if it is omitted) and returns an XMLList containing the results in order.

**Semantics**

When the processingInstruction method is called on an XMLList object \( l \) with optional parameter \text{name}, the following steps are taken:

1. If \text{name} is not specified, let \text{name} = ""
2. Let \text{name} = ToXmlName\( (\text{name}) \)
3. Let \( m \) = a new XMLList with \( m.[[\text{TargetObject}]] = l \)
4. For \( i = 0 \) to \( l.[[\text{Length}]]-1 \)
   a. If \( l[i].[[\text{Class}]] == "\text{element}" \)
      i. Let \( r = l[i].\text{processingInstruction(name)} \)
      ii. If \( r.[[\text{Length}]] > 0 \), call the [[Append]] method of \( m \) with argument \( r \)
5. Return \( m \)

12.5.4.16  XMLList.prototype.propertyIsEnumerable ( P )

**Overview**
The `propertyIsEnumerable` method returns a Boolean value indicating whether the property \( P \) is accessible via the `XMLList [[Get]]` method. All properties of `XMLList` objects accessible via the `XMLList [[Get]]` method are enumerable.

**Semantics**

When the `propertyIsEnumerable` method of an `XMLList` object \( x \) is called with parameter \( P \), the following step is taken:

1. Return the result of calling the `[[HasProperty]]` method of this object with argument \( P \).

### 12.5.4.17 `XMLList.prototype.text( )`

**Overview**

The `text` method calls the `text` method of each XML object contained in this `XMLList` object and returns an `XMLList` containing the results concatenated in order.

**Semantics**

When the `text` method is called on an `XMLList` object \( l \), the following steps are taken:

1. Let \( m \) be a new `XMLList` with \( m.[[TargetObject]] = l \)
2. For \( i = 0 \) to \( l.[[Length]]-1 \):
   a. If \( i.[[Class]] == "element" \)
      i. Let \( r = i.\text{text}() \)
      ii. If \( r.[[Length]] > 0 \), call the `[[Append]]` method of \( m \) with argument \( r \)
3. Return \( m \)

### 12.5.4.18 `XMLList.prototype.toString( )`

**Overview**

The `toString` method returns a string representation of this `XMLList` object per the `ToString` conversion operator described in section 9.1.

**Semantics**

When the `toString()` method of an `XMLList` object \( l \) is called, the following step is taken:

1. Return `ToString(l)`

### 12.5.4.19 `XMLList.prototype.toXMLString( )`

**Overview**

The `toXMLString()` method returns an XML encoded string representation of this `XMLList` object per the `ToXMLString` conversion operator described in section 9.2. Unlike the `toString` method, `toXMLString` provides no special treatment for XML objects that contain only XML text nodes (i.e., primitive values). The `toXMLString` method always calls `toXMLString` on each property contained within this `XMLList` object, concatenates the results in order and returns a single string.

**Semantics**

When the `toXMLString()` method of an `XMLList` object \( l \) is called, the following step is taken:

1. Return `toXMLString(l)`

### 12.5.4.20 `XMLList.prototype.valueOf( )`

**Overview**

The `valueOf` method returns this `XMLList` object.
Semantics

When the `valueOf` method of an XMLList object `l` is called, the following step is taken:

1. Return `l`
13 Errors
E4X extends the list of errors implementations are not required to report as specified as follows:

- An implementation may define behaviour other than throwing a TypeError exception for the ToXML function when it is called with an argument of type Object.
A Optional Features (Non-normative)

This section describes optional E4X features that are not required to be conformant with E4X, but may be provided by a conforming E4X implementation. Implementations that implement these features should conform to the associated specifications provided herein.

A.1 XML Built-in Methods

An E4X implementation may add the following methods to XML objects

A.2.1 domNode( )

Overview

The domNode method returns a W3C DOM Node representation of this XML Object.

Semantics

The semantics of the domNode method are implementation dependent.

A.2.2 domNodeList( )

Overview

The domNodeList method returns a W3C DOM NodeList containing a single W3C DOM Node representation of this XML Object.

Semantics

The semantics of the domNodeList method are implementation dependent.

A.2.3 xpath ( XPathExpression )

Overview

The xpath method evaluates the XPathExpression in accordance with the W3C XPath Recommendation using this XML object as the context node. Before evaluating the XPathExpression, the xpath method sets the XPath context as follows. The context node is set to this XML object. The context position is set to 1. The context size is set to 1. The set of variable bindings is set to the empty set. The function library is set to the empty set. The set of namespaces is set according to the set of in scope namespaces on this XML object. If the XPathExpression evaluates to a node list, the xpath method returns the results as an XMLList. Otherwise, the xpath method throws an ArgumentException. For example,

```javascript
// Use an xpath expression to get all the employees named John Smith
var jim = e.xpath("//employee[name='John Smith']")
```

Semantics

When the xpath method of an XML object x is called with parameter XPathExpression it performs the following steps:

1. Let e be the result of evaluating XPathExpression
2. Let s = ToString(GetValue(XPathExpression))
3. Create an XPath context object representing the XML object x. This semantics of this step are implementation dependent.
4. Let the XPath context position be 1
5. Let the XPath context size be 1
6. Let the XPath variable bindings be {}
7. Let the XPath function library be {}
8. Let the XPath namespaces be x.[InScopeNamespaces]]
9. Let r be the result of evaluating s in accordance with the W3C XPath Recommendation
10. If r is an XPath node-set, convert it to an XMLList in an implementation dependent way and return it.
11. Throw a TypeError exception

A.2 XMLList Built-in Methods

An E4X implementation may add the following methods to XMLList objects
A.3.1 domNodeList( )

Overview
The domNodeList method returns a W3C DOM NodeList representation of this XMLList Object.

Semantics
The semantics of the domNodeList method are implementation dependent.

A.3.2 xpath ( XPathExpression )

Overview
The xpath method evaluates the XPathExpression for each XML property contained in this XMLList object and concatenates the results an XMLList containing the results concatenated in order.

Semantics
When the xpath method is called on an XMLList object \( l \) with parameter \( XPathExpression \), the following steps are taken:

1. Let \( m = \) a new XMLList with \( l[\text{TargetObject}] = \text{null} \)
2. For \( i = 0 \) to \( l[\text{Length}]-1 \)
   a. If Type(\( l[i] \)) is XML and \( l[i].[\text{Class}] == \text{"element"} \)
      i. Let \( r = l[i].xpath(XPathExpression) \)
      ii. If \( r.[\text{Length}] > 0 \), call the \([\text{Append}]\) method of \( m \) with argument \( r \)
3. Return \( m \)
Free printed copies can be ordered from:

ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

Fax: +41 22 849.60.01
Email: documents@ecma.ch

Files of this Standard can be freely downloaded from the ECMA web site (www.ecma.ch). This site gives full information on ECMA, ECMA activities, ECMA Standards and Technical Reports.
ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

See inside cover page for obtaining further soft or hard copies.