XProc 3.0: An XML Pipeline Language

Community Group Report 12 September 2022

Specification:

https://spec.xproc.org/3.0/xproc/

Editors:

Norman Walsh

Achim Berndzen

Gerrit Imsieke

Erik Siegel

Participate:

GitHub xproc/3.0-specification

Report an issue

Errata:

https://spec.xproc.org/3.0/xproc/errata.html

This document is also available in these non-normative formats: XML, PDF (A4), PDF (US Letter).

Abstract

This specification describes the syntax and semantics of *XProc 3.0: An XML Pipeline Language*, a language for describing operations to be performed on documents.

An XML Pipeline specifies a sequence of operations to be performed on documents. Pipelines generally accept documents as input and produce documents as output. Pipelines are made up of simple steps which perform atomic operations on documents and constructs such as conditionals, iterations, and exception handlers which control which steps are executed.

Status of this Document

This specification was published by the XProc Next Community Group. It is not a W3C Standard nor is it on the W3C Standards Track. Please note that under the W3C Community Contributor License Agreement (CLA) there is a limited opt-out and other conditions apply. Learn more about W3C Community and Business Groups.

If you wish to make comments regarding this document, please send them to xproc-dev@w3.org. (subscribe, archives).

This document is derived from XProc: An XML Pipeline Language published by the W3C.

Table of Contents

1. Introduction

An XML Pipeline specifies a sequence of operations to be performed on a collection of input documents. Pipelines take documents (XML, JSON, text, images, etc.) as their input and produce documents as their output.

A *pipeline* consists of steps. Like pipelines, steps take documents as their inputs and produce documents as their outputs. The inputs of a step come from the web, from the pipeline document, from the inputs to the pipeline itself, or from the outputs of other steps in the pipeline. The outputs from a step are consumed by other steps, are outputs of the pipeline as a whole, or are discarded.

There are two kinds of steps: *atomic steps* and *compound steps*. Atomic steps carry out a single operation and have no substructure as far as the pipeline is concerned. Compound steps control the execution of other steps, which they include in the form of one or more subpipelines.

[Steps 3.0] defines a standard library of steps. Pipeline implementations **may** support additional types of steps as well.

The media type for pipeline documents is `application/xml`. Often, pipeline documents are identified by the extension `.xpl`.

In this specification the words **must**, **must not**, **should**, **should not**, **may** and **recommended** are to be interpreted as described in [RFC 2119].

1.1. Pipeline examples

Figure 1, “A simple, linear XInclude/Validate pipeline” is a graphical representation of a simple pipeline that performs XInclude processing and validation on a document.

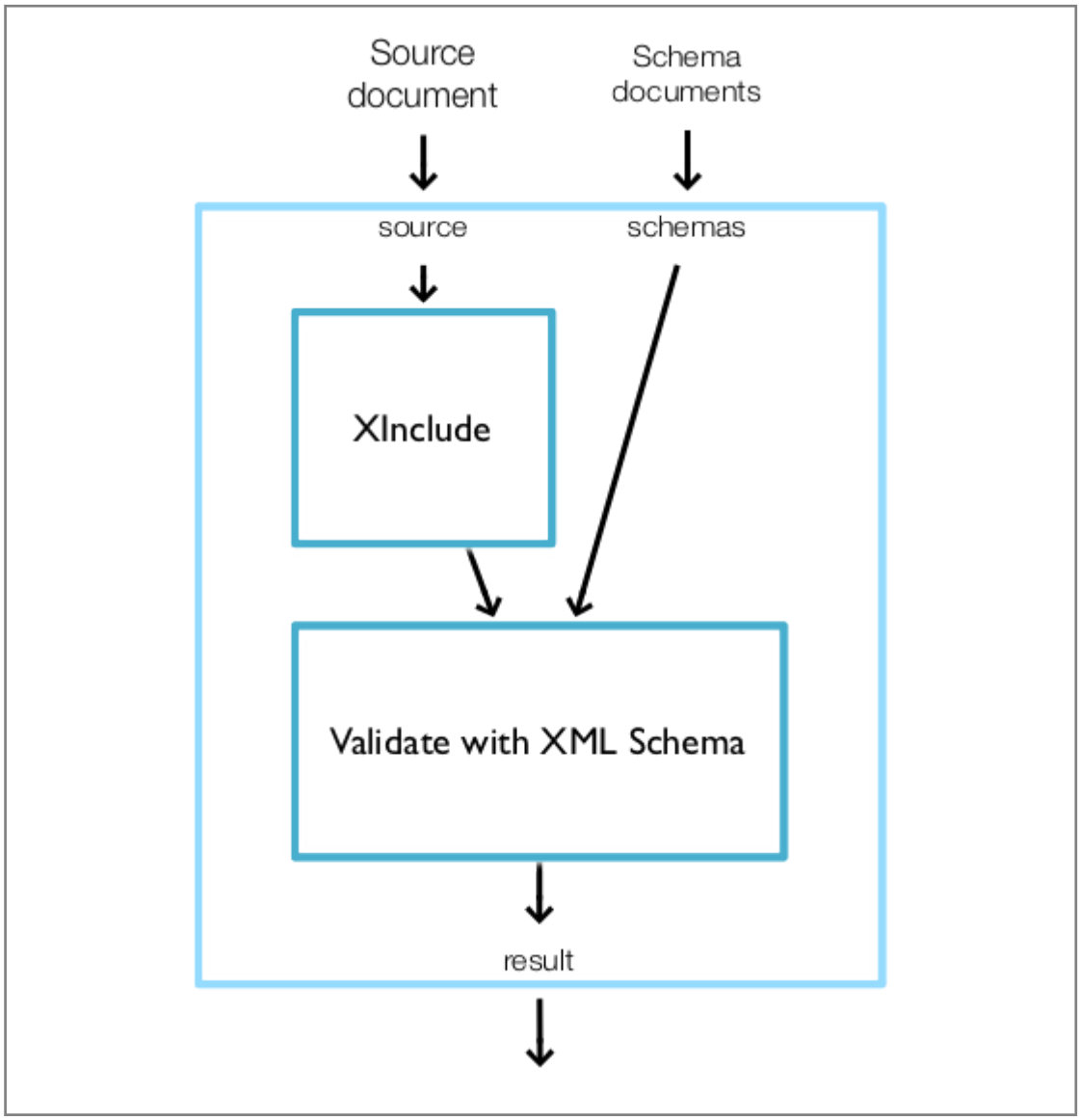

Figure 1. A simple, linear XInclude/Validate pipeline

This is a pipeline that consists of two atomic steps, XInclude and Validate with XML Schema. The pipeline itself has two inputs, “source” (a source document) and “schemas” (a sequence of W3C XML Schemas). The XInclude step reads the pipeline input “source” and produces a result document. The Validate with XML Schema step reads the pipeline input “schemas” and the result of the XInclude step and produces its own result document. The result of the validation, “result”, is the result of the pipeline. (For consistency across the step vocabulary, the standard input is usually named “source” and the standard output is usually named “result”.)

The pipeline document determines how the steps are connected together inside the pipeline, that is, how the output of one step becomes the input of another.

The pipeline document for this pipeline is shown in Example 1, “A simple, linear XInclude/Validate pipeline”.

Example 1. A simple, linear XInclude/Validate pipeline

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                name="xinclude-and-validate"
                version="3.0">
  <p:input port="source" primary="true"/>
  <p:input port="schemas" sequence="true"/>
  <p:output port="result">
    <p:pipe step="validated" port="result"/>
  </p:output>

  <p:xinclude name="included">
    <p:with-input port="source">
      <p:pipe step="xinclude-and-validate" port="source"/>
    </p:with-input>
  </p:xinclude>

  <p:validate-with-xml-schema name="validated">
    <p:with-input port="source">
      <p:pipe step="included" port="result"/>
    </p:with-input>
    <p:with-input port="schema">
      <p:pipe step="xinclude-and-validate" port="schemas"/>
    </p:with-input>
  </p:validate-with-xml-schema>
</p:declare-step>
```

Example 1, “A simple, linear XInclude/Validate pipeline” is very verbose. It makes all of the connections seen in the figure explicit. In practice, pipelines do not have to be this verbose. By default, where inputs and outputs are connected between sequential sibling steps, they do not have to be made explicit.

The same pipeline, using XProc defaults, is shown in Example 2, “A simple, linear XInclude/Validate pipeline (simplified)”.

Example 2. A simple, linear XInclude/Validate pipeline (simplified)

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                name="xinclude-and-validate"
                version="3.0">
  <p:input port="source" primary="true"/>
  <p:input port="schemas" sequence="true"/>
  <p:output port="result"/>

  <p:xinclude/>

  <p:validate-with-xml-schema>
    <p:with-input port="schema">
      <p:pipe step="xinclude-and-validate" port="schemas"/>
    </p:with-input>
  </p:validate-with-xml-schema>
</p:declare-step>
```

Figure 2, “A validate and transform pipeline” is a more complex example: it performs schema validation with an appropriate schema and then styles the validated document.

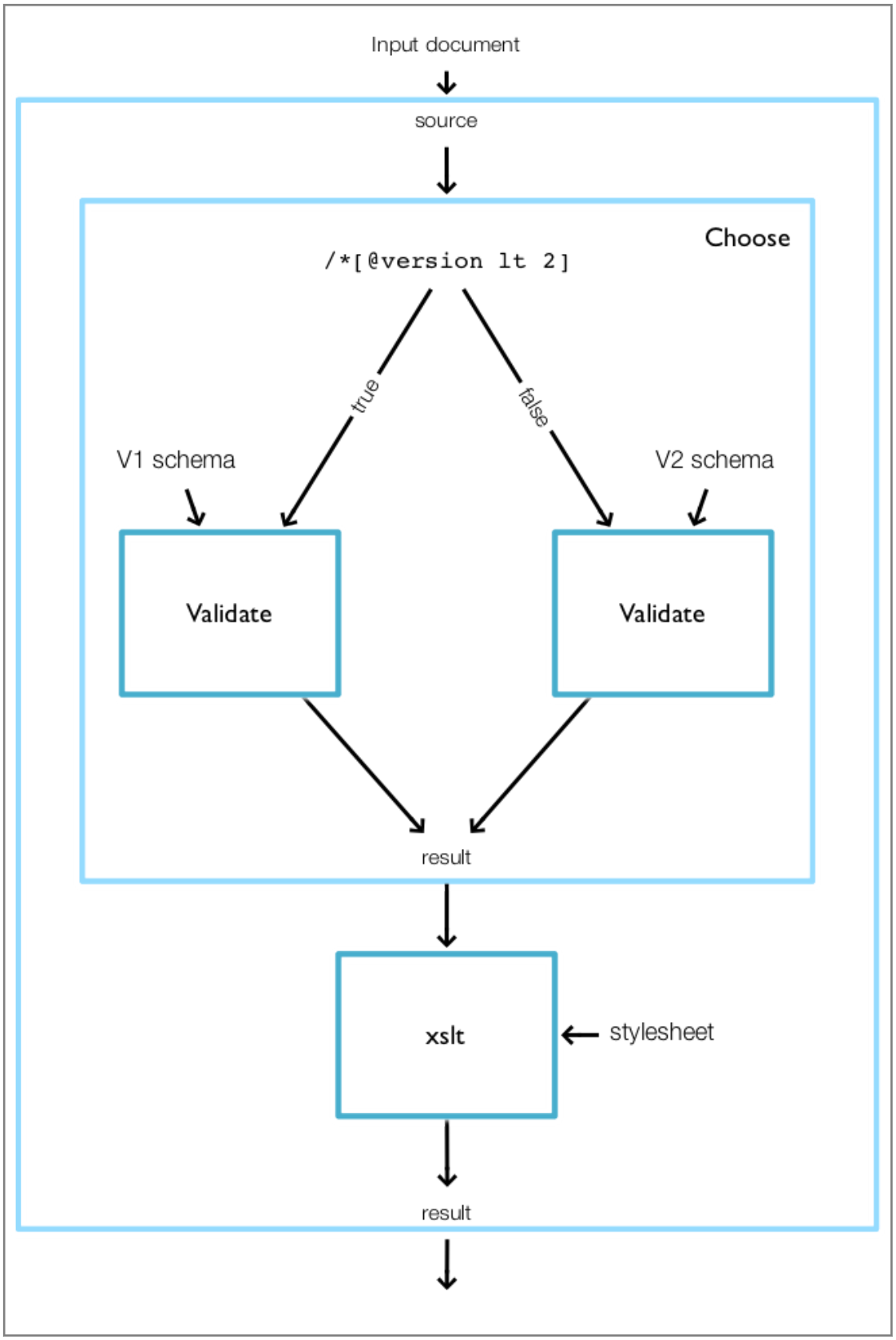

Figure 2. A validate and transform pipeline

The heart of this example is the conditional. The “choose” step evaluates an XPath expression over a test document. Based on the result of that expression, one or another branch is run. In this example, each branch consists of a single validate step.

Example 3. A validate and transform pipeline

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                name="xinclude-and-validate"
                version="3.0">
  <p:input port="source"/>
  <p:input port="schemas" sequence="true"/>
  <p:output port="result"/>

  <p:choose>
    <p:when test="/*[@version &lt; 2.0]">
      <p:validate-with-xml-schema>
        <p:with-input port="schema" href="v1schema.xsd"/>
      </p:validate-with-xml-schema>
    </p:when>

    <p:otherwise>
      <p:validate-with-xml-schema>
        <p:with-input port="schema" href="v2schema.xsd"/>
      </p:validate-with-xml-schema>
    </p:otherwise>
  </p:choose>

  <p:xslt>
    <p:with-input port="stylesheet" href="stylesheet.xsl"/>
  </p:xslt>
</p:declare-step>
```

This example, like the preceding, relies on XProc defaults for simplicity. It is always valid to write the fully explicit form if you prefer. This example also takes advantage of using the `href` attribute directly on `p:with-input` as a shortcut for the `p:document` connection.

2. Pipeline Concepts

[Definition: A *pipeline* is a set of connected steps, with outputs of one step flowing into inputs of another.] A pipeline is itself a *step* and must satisfy the constraints on steps. Connections between steps occur where the input of one step is connected to the output of another.

The result of evaluating a pipeline (or *subpipeline*) is the result of evaluating the steps that it contains, in an order consistent with the connections between them. A pipeline must behave as if it evaluated each step each time it is encountered. Unless otherwise indicated, implementations **must not** assume that steps are functional (that is, that their outputs depend only on their inputs and *options*) or side-effect free.

The pattern of connections between steps will not always completely determine their order of evaluation. The evaluation order of steps not connected to one another is *implementation-dependent*.

2.1. Steps

[Definition: A *step* is the basic computational unit of a pipeline.] A typical step has inputs, from which it receives documents to process, outputs, to which it sends result documents, and options which influence its behavior.

There are two kinds of steps: atomic and compound:

- An atomic step is a step that performs a unit of processing on its input, such as validation or transformation, and has no internal subpipeline. Atomic steps carry out fundamental operations and can perform arbitrary amounts of computation, but they are indivisible.

  There are many *types* of atomic steps. The standard library of atomic steps is described in [Steps 3.0], but implementations **may** provide others as well. It is *implementation-defined* what additional step types, if any, are provided. Each use, or instance, of an atomic step invokes the processing defined by that type of step. A pipeline may contain instances of many types of steps and many instances of the same type of step.

- Compound steps, on the other hand, control and organize the flow of documents through a pipeline, providing familiar programming language functionality such as conditionals, iterators and exception handling. They contain other steps, whose evaluation they control.

[Definition: A *compound step* is a step that contains one or more *subpipelines*.] That is, a compound step differs from an atomic step in that its semantics are at least partially determined by the steps that it contains.

Compound steps either directly contain a single subpipeline or contain several subpipelines and select one or more to evaluate dynamically. In the latter case, alternate subpipelines are identified by non-step wrapper elements that each contain a single subpipeline.

[Definition: A *container* is either a compound step or one of the non-step wrapper elements in a compound step that contains several subpipelines.] [Definition: The steps that occur directly within a container are called that step's *contained steps*. In other words, "container" and "contained steps" are inverse relationships.] [Definition: The *ancestors* of a step, if it has any, are its *container* and the ancestors of its container.]

[Definition: Sibling steps and variables (and the connections between them) form a *subpipeline*.] [Definition: The *last step* in a subpipeline is its last step in document order.]

```
subpipeline = (p:variable | p:for-each | p:viewport | p:choose | p:if |
               p:group | p:try | p:standard-step | pfx:user-pipeline)+
```

> **Note**
>
> When a user-defined pipeline is invoked, (identified with *`pfx:user-pipeline`* in the preceding syntax summary) it appears as an atomic step. A pipeline *declaration* may contain a subpipeline, but the *invocation* of that pipeline is atomic and does not contain a subpipeline.

Steps have “ports” into which inputs and outputs are connected. Each step has a number of input ports and a number of output ports; a step can have zero input ports and/or zero output ports. The names of all ports on each step must be unique on that step (you can't have two input ports named “source”, nor can you have an input port named “schema” and an output port named “schema”).

A step may have zero or more options, all with unique names.

2.1.1. Step names

All of the different instances of steps (atomic or compound) in a pipeline can be distinguished from one another by *name*. Names can be specified using the (optional) `name` attribute. A specified step name **must** be unique within its scope, see Section 14.2, “Scoping of Names”.

The main purpose of step names in a pipeline is to *explicitly* connect to port(s) of another step. This applies to `p:input`, `p:output`, `p:with-input`, `p:with-option` and `p:variable`, using either a `p:pipe` child element or a `pipe` attribute.

The following example uses the step names provided by the `name` attributes to explicitly connect ports, using `p:pipe` child elements. The document on the `extra` port of the step gets an additional attribute and is subsequently inserted into the document on the `source` port.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0"
    name="main-step">

  <p:input port="source" primary="true"/>
  <p:input port="extra"/>
  <p:output port="result" primary="true"/>

  <p:add-attribute attribute-name="timestamp"
                   attribute-value="{current-dateTime()}"
      name="add-timestamp">
    <p:with-input port="source">
      <p:pipe step="main-step" port="extra"/>
    </p:with-input>
  </p:add-attribute>

  <p:insert match="/*" position="first-child">
    <p:with-input port="source">
      <p:pipe step="main-step" port="source"/>
    </p:with-input>
    <p:with-input port="insertion">
      <p:pipe step="add-timestamp" port="result"/>
    </p:with-input>
  </p:insert>

</p:declare-step>
```

Alternatively, using `pipe` attributes to connect the ports, you could write this as:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0"
    name="main-step">

  <p:input port="source" primary="true"/>
  <p:input port="extra"/>
  <p:output port="result" primary="true"/>

  <p:add-attribute attribute-name="timestamp"
                   attribute-value="{current-dateTime()}"
      name="add-timestamp">
    <p:with-input port="source" pipe="extra@main-step"/>
  </p:add-attribute>

  <p:insert match="/*" position="first-child">
    <p:with-input port="source" pipe="source@main-step"/>
    <p:with-input port="insertion" pipe="result@add-timestamp"/>
  </p:insert>

</p:declare-step>
```

If the pipeline author does not provide an explicit name using the `name` attribute, the processor manufactures a default name. All default names are of the form "`!1.`*m*`.`*n*`…`" where "*m*" is the position (in the sense of counting sibling elements) of the step's highest ancestor element within the pipeline document or library which contains it, "*n*" is the position of the next-highest ancestor, and so on, including all of the elements in the pipeline document (that were not *effectively excluded*). For example, consider the pipeline in Example 3, "A validate and transform pipeline". The `p:declare-step` step has no name, so it gets the default name "`!1`"; the `p:choose` gets the name "`!1.1`"; the first `p:when` gets the name "`!1.1.1`"; the `p:otherwise` gets the name "`!1.1.2`", etc. If the `p:choose` had a name, it would not have received a default name, but it would still have been counted and its first `p:when` would still have been "`!1.1.1`".

Providing every step in the pipeline with an interoperable name has several benefits:

1. It allows implementers to refer to all steps in an interoperable fashion, for example, in error messages.

2. Pragmatically, we say that *readable ports* are identified by a step name/port name pair. By manufacturing names for otherwise anonymous steps, we include implicit connections without changing our model.

In a valid pipeline that runs successfully to completion, the manufactured names aren't visible (except perhaps in debugging or logging output).

> **Note**
>
> The format for defaulted names does not conform to the requirements of an NCName. This is an explicit design decision; it prevents pipelines from using the defaulted names on `p:pipe` elements. If an explicit connection requires a step name, the pipeline author must name the step.

2.1.2. Step types

A step can have a *type*. Step types are specified using the `type` attribute of the `p:declare-step` element. Step types are used as the name of the element by which the step is invoked. A specified step type **must** be unique within its scope, see Section 14.2, “Scoping of Names”.

Step types are QNames and **must** be in namespace with a non-null namespace URI. Steps in the XProc standard and additional step libraries all have types in the XProc namespace: `http://www.w3.org/ns/xproc`. When providing your own steps with a type, which is necessary to use/invoke them in another step, a different (non-null) namespace must be used.

Step types play an important role in the modularization of pipelines. They allow steps to be re-used. The following example defines a local step with type `mysteps:add-timestamp-attribute` and subsequently uses this twice somewhere inside its main step’s pipeline:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0"
  xmlns:mysteps="http://.../ns/mysteps">

  <p:input port="source"/>
  <p:output port="result"/>

  <p:declare-step type="mysteps:add-timestamp-attribute">
    <p:input port="source"/>
    <p:output port="result"/>
    <p:add-attribute attribute-name="timestamp"
                     attribute-value="{current-dateTime()}"/>
  </p:declare-step>

  ...
  <mysteps:add-timestamp-attribute/>
  ...
  <mysteps:add-timestamp-attribute/>
  ...

</p:declare-step>
```

Another way of doing this would be to isolate the `mysteps:add-timestamp-attribute` step as a separate, stand-alone, pipeline:

```
<p:declare-step type="mysteps:add-timestamp-attribute" version="3.0"
                xmlns:p="http://www.w3.org/ns/xproc"
                xmlns:mysteps="http://.../ns/mysteps">
  <p:input port="source"/>
  <p:output port="result"/>
  <p:add-attribute attribute-name="timestamp"
                   attribute-value="{current-dateTime()}"/>
</p:declare-step>
```

Assuming this is stored in a file called `add-timestamp-attribute.xpl`, you can now use the `p:import` element to get it on board:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="3.0"
  xmlns:mysteps="http://.../ns/mysteps">

  <p:import href="add-timestamp-attribute.xpl"/>

  <p:input port="source"/>
  <p:output port="result"/>

  ...
  <mysteps:add-timestamp-attribute/>
  ...

</p:declare-step>
```

Yet another way of achieving the same result would be to add the `mysteps:add-timestamp-attribute` step to an XProc *library*, using the `p:library` root element:

```
<p:library version="3.0" xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:mysteps="http://.../ns/mysteps">

  <p:declare-step type="mysteps:add-timestamp-attribute">
    <p:input port="source"/>
    <p:output port="result"/>
    <p:add-attribute attribute-name="timestamp"
                     attribute-value="{current-dateTime()}"/>
  </p:declare-step>

  ... more library steps

</p:library>
```

Importing a library is also done with the `p:import` element.

3. Documents

An XProc pipeline processes documents. [Definition: A *document* is a *representation* and its *document properties*.]. [Definition: A *representation* is a data structure used by an XProc processor to refer to the actual document content.]

An output port may have several connections. In this case the document(s) that appear on that port are sent to each of the connections. In principle, a distinct copy of each document is sent to each connection. Critically, any changes made to one copy **must not** be visible in any other copy. In the interest of efficiency, if an implementation can isolate such changes, it is not required to make actual copies.

3.1. Document Properties

Documents have associated with them a set of properties. [Definition: The *document properties* are key/value pairs; they are exposed to the XProc pipeline as a map (`map(xs:QName, item()*)`).]

Several properties are defined by this specification:

`content-type`
: The value of the "`content-type`" property identifies the media type ([RFC 2046]) of the representation. The "`content-type`" **must** always be present. The processor is responsible for assuring that the `content-type` property matches the content type of each document produced on every output port.

`base-uri`
: The value of the "`base-uri`" property identifies the base URI of the document; it will only be present if the document has a base URI. For XML documents, HTML documents, and text documents, the value of "`base-uri`" is the base URI property of the document node. For other document types it is a property the processor keeps track of. If no such key is present, the document has no base URI.

`serialization`
: The value of the (optional) "`serialization`" property holds serialization properties for the document. If present, it's value **must** be of type `map(xs:QName, item()*)`. It is a *dynamic error* (`err:XD0070`) if a value is assigned to the `serialization` document property that cannot be converted into `map(xs:QName, item()*)` according to the rules in Implicit Casting. Serialization properties control XML serialization as defined by [Serialization]. See also Section 16.3.1, "Serialization parameters".

> Some steps, like `p:xslt` and `p:xquery`, can specify serialization properties (for instance using an XSLT `xsl:output` element). If this is the case, the specified serialization properties **should** be returned in the result document(s) `serialization` property, as an appropriate serialization properties map.
>
> If a step serializes a document whose document properties contain a `serialization` property, it **must** use these serialization properties. If the step itself allows specification of serialization properties (usually by a `serialization` option), both sets of serialization properties are merged. Serialization properties specified on the step itself have precedence over serialization properties specified with the `serialization` document property.

Other property keys may also be present, including user defined properties.

Steps are responsible for describing how document properties are transformed as documents flow through them. Many steps claim that the specified properties are preserved. Generally, it is the responsibility of the pipeline author to determine when this is inappropriate and take corrective action. However, it is the responsibility of the pipeline processor to assure that the `content-type` property is correct. If a step transforms a document in a manner that is inconsistent with the `content-type` property (accepting an XML document on the source port but producing a text document on the result, for example), the processor must assure that the `content-type` property is appropriate. If a step changes the `content-type` in this way, it **must** also remove the `serialization` property.

3.2. Document Types

XProc 3.0 has been designed to make it possible to process any kind of document. Each document has a representation in the [XQuery and XPath Data Model 3.1]. This is necessary so that any kind of document can be passed as an argument to XPath functions, such as `p:document-properties`. Practically speaking, there are five kinds of documents:

1. XML documents
2. HTML documents
3. Text documents

4. JSON documents
5. Other documents

3.2.1. XML Documents

Representations of XML documents are general instances of the XDM. They are documents that contain a mixture of other node types (elements, text, comments, and processing instructions). This definition is intentionally broader than the definition of a well-formed XML document because it is often convenient for intermediate stages in a pipeline to produce more-or-less arbitrary fragments of XML that can be combined together by later stages. XML documents are identified by an XML media type. [Definition: The "`application/xml`" and "`text/xml`" media types and all media types of the form "`something/something+xml`" (except for "`application/xhtml+xml`" which is explicitly an *HTML media type*) are *XML media types*.]

In order to be consistent with the XPath data model, all general and external parsed entities **must** be fully expanded in XML documents; they **must not** contain any representation of [Infoset] `[unexpanded entity reference information items]`.

The level of support for typed values in XDM instances in an XProc pipeline is *implementation-defined*.

When an XML document is serialized, it **should** be serialized using the XML serializer (see [Serialization]) by default.

3.2.2. HTML Documents

Representations of HTML documents are general instances of the XDM. Within XProc, they are XML documents. HTML documents are identified by an HTML media type. [Definition: The "`text/html`" and "`application/xhtml+xml`" media types are *HTML media types*.]

The distinction between XML documents and HTML documents is apparent in two places:

1. When an HTML document is *parsed*, for example when it is the result of querying a web service or is loaded from a file on disk, an HTML parser **must** be used. An HTML parser will construct a balanced tree even if the HTML document would not be seen as well-formed XML if it was parsed by an XML parser. An HTML parser may also add elements not found in the original (for example table body elements inside tables).

> **Note**
>
> The HTML parsing rules only apply when the content is parsed. HTML content in an unencoded `p:inline` must be well-formed XML (because it is literally in the pipeline) and will not be transformed in any way.

2. When an HTML document is serialized, it **should** be serialized using the HTML serializer for documents with media type "`text/html`" and the XHTML serializer for those with media type "`application/xhtml+xml`" (see [Serialization]) by default.

3.2.3. Text Documents

Text documents are identified by a text media type. [Definition: Media types of the form "`text/`*`something`*" are *text media types* with the exception of "`text/xml`" which is an XML media type, and "`text/html`" which is an HTML media type. Additionally the media types "`application/javascript`", "`application/relax-ng-compact-syntax`", and "`application/xquery`" are also text media types.] It is *implementation-defined* whether other media types not mentioned in this document are treated as text media types as well. A text document is represented by a document node containing a single text node or by an empty document node (for empty text documents).

When a text document is serialized, it **should** be serialized using the Text serializer (see [Serialization]) by default.

3.2.4. JSON Documents

Representations of JSON documents are instances of the XDM. They are maps, arrays, or atomic values. JSON documents are identified by a JSON media type. [Definition: The "`application/json`" media type and all media types of the form "`application/`*`something`*`+json`" are *JSON media types*.]

When a JSON document is serialized, it **should** be serialized using the JSON serializer (see [Serialization]) by default.

3.2.5. Other documents

Representations of other kinds of documents are empty XDM documents. The *underlying* representations of other kinds of documents are *implementation-dependent*. Other kinds of documents are identified by media types that are not *XML media types*, *HTML media types*, *text media types*, or *JSON media types*.

Serialization of other kinds of documents is *implementation-defined*. The stored sequence of octets **should** be consistent with the media type: an `image/png` image should be a PNG image, etc.

3.3. Creating documents from XDM step results

Some steps like `p:xslt`, `p:xquery` etc. create a sequence of new XDM instances. The same is true for the result of a `select` expression on `p:with-input`. Values in such a sequence can be of any XDM type (except attribute or function). Every item in such a sequence is converted into a *separate* document that will appear on the output port of that particular step or as the result of the `p:with-input`. The following rules apply to each of the items in the output sequence:

- If the item is a text node, it is wrapped in a document node and the document's content-type is `text/plain`.
- If the item is an element, comment or processing-instruction node, a document node is wrapped around the node and the document's content-type is set to `application/xml`.
- If the item is a document node, content-type "application/xml" is used.

- If the item is a `map`, `array` or any atomic value, content-type `application/json` is used.

> **Note**
>
> Setting the content-type to `application/json` for *any* map, array or atomic value means that a document with content-type `application/json` is *not* guaranteed serializable using the `json` serialization method. For instance, a map with values that contain sequences cannot be serialized.

3.4. Specifying content types

In some contexts (step inputs, and step outputs, for example), XProc allows the pipeline author to specify a list of content types to identify what kinds of documents are allowed. Each content type in this list must have one of the following forms:

- A fully qualified type of the form "*`type/subtype+ext`*" where "`+ext`" is optional and any of *`type`*, *`subtype`*, and *`ext`* can be specified as "`*`" meaning "any". For example: `text/plain` (only plain text documents), `text/*` (any "text" content type), `*/*+xml` (any "+xml" content type), and `*/*` (any content type).
- A fully qualified type preceded by a minus sign ("-") indicates that the specified type is forbidden. For example: `-image/svg` forbids SVG images, `-text/*` forbids "text" content types, and `-text/html` forbids HTML documents.
- A single token (without a "/"), is considered a shortcut form. The following shortcuts **must** be supported by the processor:

  `xml`
  : Expands to "`application/xml text/xml */*+xml -application/xhtml+xml`".

  `html`
  : Expands to "`text/html application/xhtml+xml`".

text
: Expands to: "`text/* -text/html -text/xml`".

json
: Expands to "`application/json`".

any
: Expands to "`*/*`".

It is a *dynamic error* (`err:XD0079`) if a supplied content-type is not a valid media type of the form "*`type/subtype+ext`*" or "*`type/subtype`*". It is *implementation-defined* if a processor accepts any other content type shortcuts. It is a *static error* (`err:XS0111`) if an unrecognized content type shortcut is specified.

To determine if a document is acceptable, the (expanded) list of content types is considered from left to right. If the actual content type matches an acceptable content type, the document is acceptable. If it matches a forbidden content type, then it is not. A content type that isn't matched is ignored. The document is considered acceptable if and only if it matches at least one acceptable content type and the last content type that matched was not forbidden.

For example: a document with the content type "`image/svg`" is acceptable if the content type list expands to "`image/* application/xml`" but it is not acceptable if the content type list expands to "`image/* -image/svg`". (Note that order matters; the document would be considered acceptable if the content type list expands to "`-image/svg image/*`".)

In the particular case of shortcut values, note that "`application/xhtml+xml`" is acceptable if the content type list is "`xml html`" but not if it is "`html xml`".

It is a *dynamic error* (`err:XD0038`) if an input document arrives on a port and it does not match the allowed content types.

4. Inputs and Outputs

Most steps have one or more inputs and one or more outputs. Figure 3, "An atomic step" illustrates symbolically an *atomic step* with two inputs and one output.

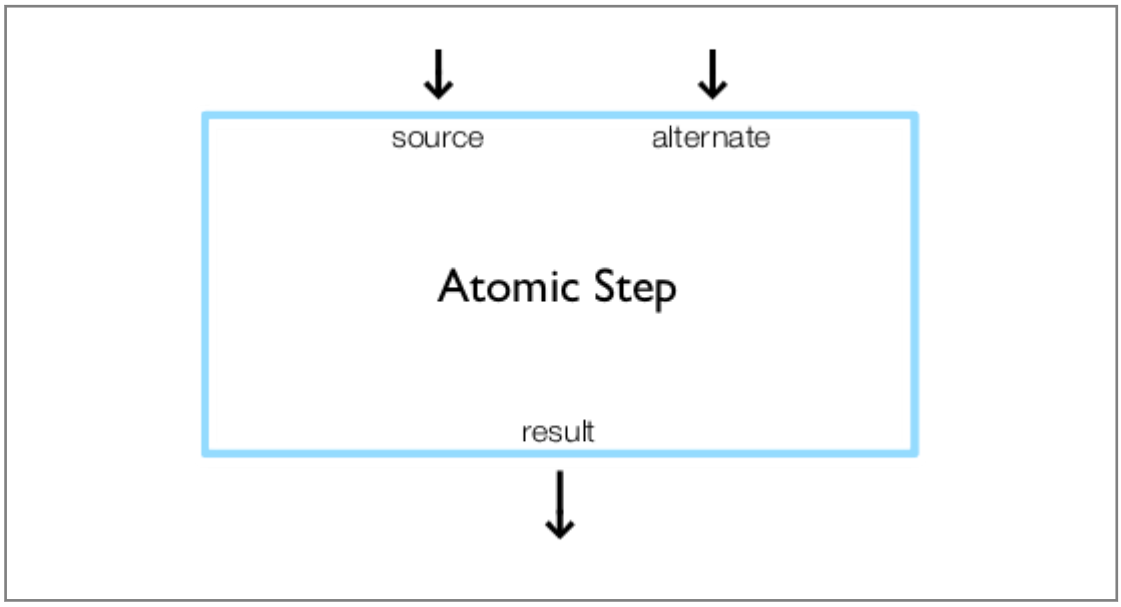

Figure 3. An atomic step

All atomic steps are defined by a `p:declare-step`. The declaration of an atomic step type defines the input ports, output ports, and options of all steps of that type. For example, every `p:validate-with-xml-schema` step has two inputs, named "`source`" and "`schema`", one output named "`result`", and the same set of options.

Like atomic steps, top level, user-defined pipelines also have declarations.

The situation is slightly more complicated for the other compound steps because they don't have separate declarations; each instance of the compound step serves as its own declaration. On these compound steps, the number and names of the outputs can be different on each instance of the step.

Figure 4, "A compound step" illustrates symbolically a compound step with a subpipeline with one output. As you can see from the diagram, the output from the compound step comes from one of the outputs of the subpipeline within the step.

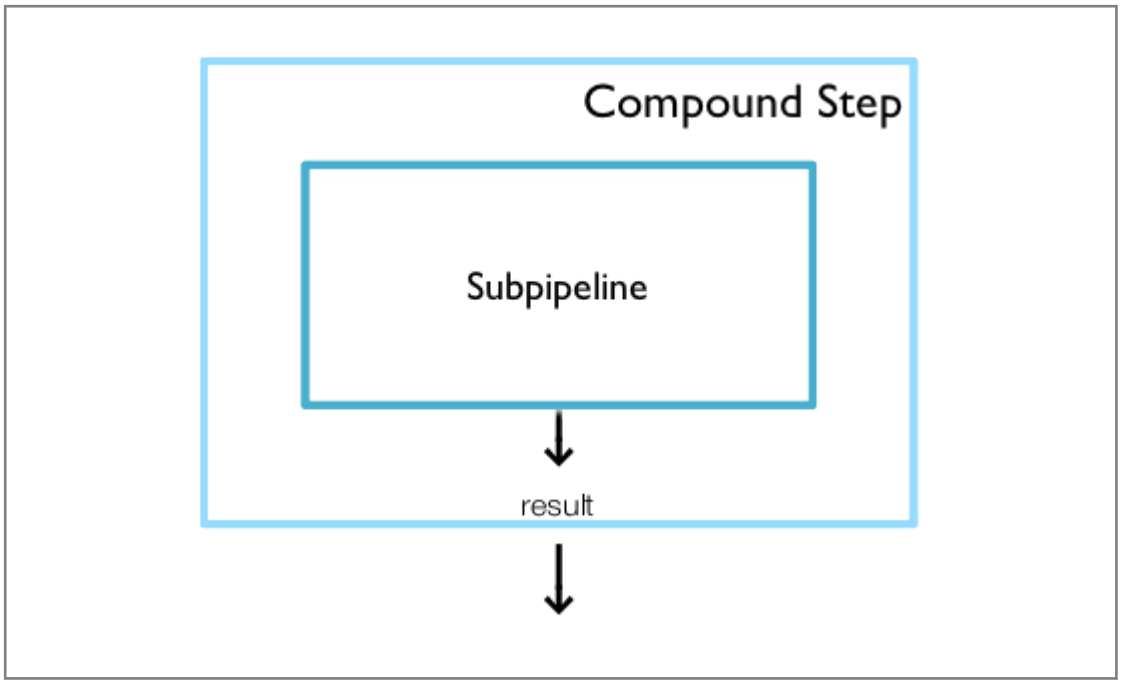

Figure 4. A compound step

[Definition: The input ports declared on a step are its *declared inputs*.] [Definition: The output ports declared on a step are its *declared outputs*.] When a step is used in a pipeline, it is connected to other steps through its inputs and outputs.

[Definition: The *compound steps* `p:for-each` and `p:viewport` each declare a single primary input without a port name. Such an input is called an *anonymous input*.]

When a step is used, all of the *declared inputs* of the step **must** be connected. Each connection binds the input to a data source (see Section 6, "Connections"). It is a *static error* (`err:XS0003`) if any declared input is not connected.

The *declared outputs* of a step are only connected when they are used by another step or expression. Any documents produced on an unconnected output port are discarded.

Primary input and primary output ports may be implicitly connected if no explicit connection is given, see Section 5, "Primary Inputs and Outputs".

Output ports on compound steps have a dual nature: from the perspective of the compound step's siblings, its outputs are just ordinary outputs and can be connected

the same as other *declared outputs*. From the perspective of the subpipeline inside the compound step, they behave like inputs and can be connected just like other inputs.

Within a compound step, the *declared outputs* of the step can be connected to any of the various available outputs of *contained steps* as well as other data sources (see Section 6, "Connections"). If a (non-primary) output port of a compound step is left unconnected, it produces an empty sequence of documents from the perspective of its siblings.

Each input and output on a step is declared to accept or produce either a single document or a sequence of documents. It *is not* an error to connect a port that is declared to produce a sequence of documents to a port that is declared to accept only a single document. It is, however, an error if the former step does not produce exactly one document at run time.

It is also not an error to connect a port that is declared to produce a single document to a port that is declared to accept a sequence. A single document is the same as a sequence of one document.

An output port may have more than one connection: it may be connected to more than one input port, more than one of its container's output ports, or both. At runtime this will result in the outputs being sent to each of those places.

[Definition: The *signature* of a step is the set of inputs, outputs, and options that it is declared to accept.] The declaration for a step provides a fixed signature which all its instances share.

> **Note**
>
> Within the context of what can be defined by XProc pipelines, step signatures are fixed and shared by all instances. There is no mechanism for a pipeline author to declare that an atomic step has a signature that varies. However, implementors may provide such mechanisms and other specifications may depend upon them. Such steps are "magic" and XProc 3.0 makes no effort to provide a mechanism to define them.

[Definition: A step *matches* its signature if and only if it specifies an input for each declared input, it specifies no inputs that are not declared, it specifies an option for each option that is declared to be required, and it specifies no options that are not declared.] In other words, every input and required option **must** be specified and only inputs and options that are declared **may** be specified. Options that aren't required do not have to be specified.

Steps **may** also produce error, warning, and informative messages. These messages are captured and provided on the `error` port inside of a `p:catch`. Outside of a try/catch, the disposition of error messages is *implementation-dependent*.

How inputs are connected to documents outside the pipeline is *implementation-defined*.

How pipeline outputs are connected to documents outside the pipeline is *implementation-defined*.

Input ports **may** specify a content type, or list of content types, that they accept, see Section 3.4, "Specifying content types".

4.1. External Documents

It's common for some of the documents used in processing a pipeline to be read from URIs. Sometimes this occurs directly, for example with a `p:document` element. Sometimes it occurs indirectly, for example if an implementation allows the URI of a pipeline input to be specified on the command line or if an `p:xslt` step encounters an `xsl:import` in the stylesheet that it is processing. It's also common for some of the documents produced in processing a pipeline to be written to locations which have, or at least could have, a URI.

The process of dereferencing a URI to retrieve a document is often more interesting than it seems at first. On the web, it may involve caches, proxies, and various forms of indirection. Resolving a URI locally may involve resolvers of various sorts and possibly appeal to *implementation-dependent* mechanisms such as catalog files.

In XProc, the situation is made even more interesting by the fact that many intermediate results produced by steps in the pipeline have base URIs. Whether (and

when and how) or not the intermediate results that pass between steps are ever written to a filesystem is *implementation-dependent*.

In Version 3.0 of XProc, how (or if) implementers provide local resolution mechanisms and how (or if) they provide access to intermediate results by URI is *implementation-defined*.

Version 3.0 of XProc does not require implementations to guarantee that multiple attempts to dereference the same URI always produce the same results.

> **Note**
>
> On the one hand, this is a somewhat unsatisfying state of affairs because it leaves room for interoperability problems. On the other, it is not expected to cause such problems very often in practice.
>
> If these problems arise in practice, implementers are encouraged to use the existing extension mechanisms to give users the control needed to circumvent them. Should such mechanisms become widespread, a standard mechanism could be added in some future version of the language.

5. Primary Inputs and Outputs

Each step may have one input port designated as the primary input port and one output port designated as the primary output port.

[Definition: If a step has an input port which is explicitly marked "`primary='true'`", or if it has exactly one document input port and that port is *not* explicitly marked "`primary='false'`", then that input port is the *primary input port* of the step.] If a step has a single input port and that port is explicitly marked "`primary='false'`", or if a step has more than one input port and none is explicitly marked as the primary, then the primary input port of that step is undefined. A step can have at most one primary input port.

[Definition: If a step has an output port which is explicitly marked "`primary='true'`", or if it has exactly one document output port and that port is *not* explicitly marked "`primary='false'`", then that output port is the *primary output port* of the step.] If a step has a single output port and that port is explicitly marked "`primary='false'`", or if a step has more than one output port and none is explicitly marked as the primary, then the primary output port of that step is undefined. A step can have at most one primary output port.

The special significance of primary input and output ports is that they are connected automatically by the processor if no explicit connection is given. Generally speaking, if two steps appear sequentially in a subpipeline, then the primary output of the first step will automatically be connected to the primary input of the second.

Additionally, if a container, that can have declared outputs, has no declared outputs and the *last step* in its subpipeline has an unconnected primary output, then an implicit primary output port will be added to the compound step (and consequently the last step's primary output will be connected to it). This implicit output port has no name. It inherits the `sequence` and the `content-types` properties of the port connected to it. This rule does not apply to `p:declare-step`; step declarations must provide explicit names for all of their outputs.

6. Connections

Steps are connected together by their input ports, output ports, and bindings to variables and options. Variables and options also behave something like steps, connected together by the input on which they receive their context and by references to them by name elsewhere. It is a *static error* (`err:XS0001`) if there are any loops in the connections between steps, variables, and options: no step, variable, or option can be connected to itself nor can there be any sequence of connections through other steps that leads back to itself.

Consider Figure 5, "Dependencies between steps, variables, and options".

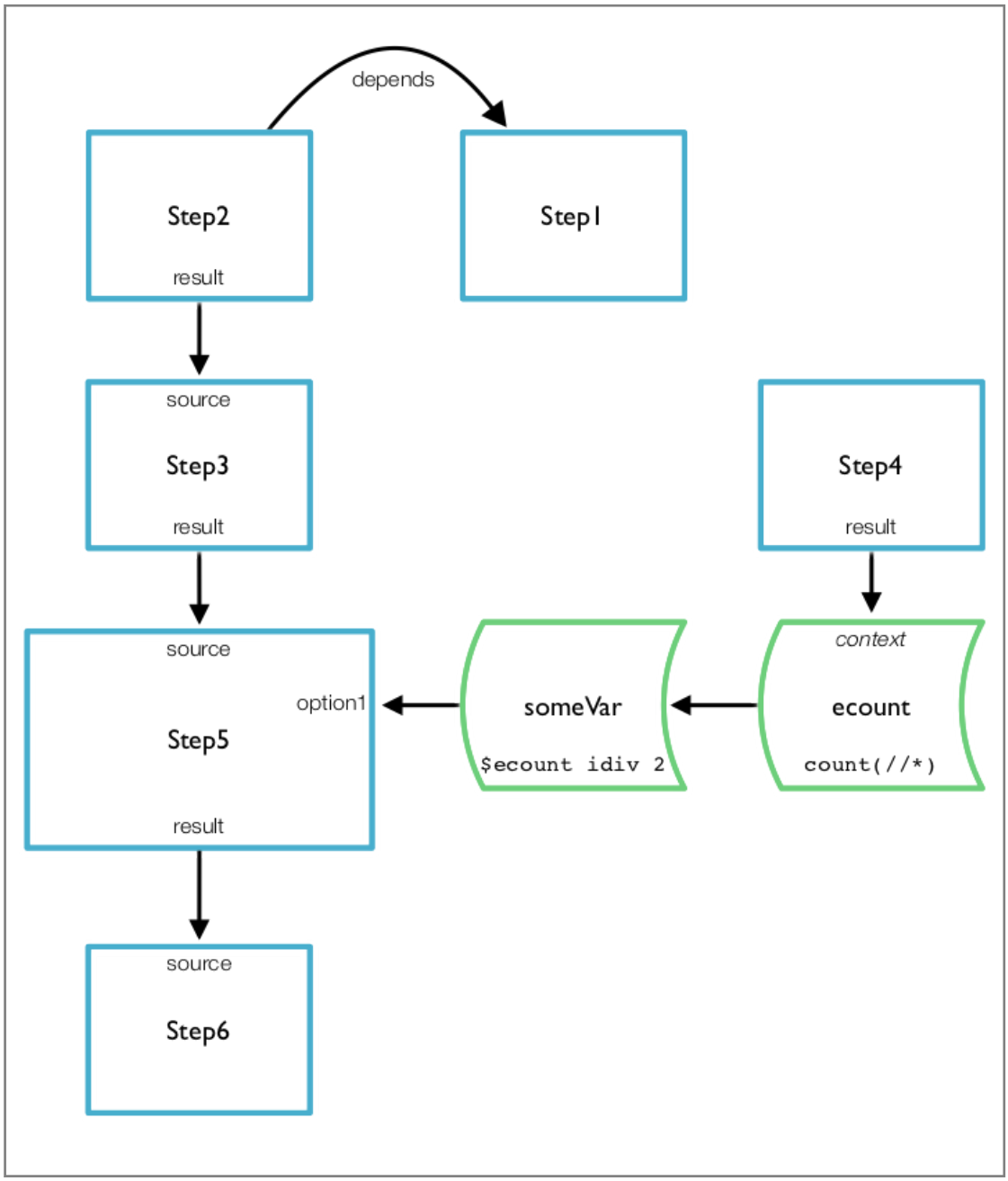

Figure 5. Dependencies between steps, variables, and options

- Step1 has no connections.
- Step2 is connected to Step1 by an explicit dependency, see Section 14.9.3, “Additional dependent connections”.

- Step3 is connected to Step2 because it reads from the output of Step2. It is also transitively connected to Step1 because Step2 is connected to it.
- Step4 has no connections. In principle, Step1 and Step4 can be evaluated in parallel or in either order.
- Step5 is connected to Step3 because it reads from the output of Step3. It is also transitively connected to Step2 and the connections that Step2 has. Step5 is also connected to Step4 because it's option "`option1`" is connected to "`someVar`" which is connected to "`ecount`" which reads its context from Step4.
- Step6 is connected to Step5 because it reads from the output of Step5. It is also transitively connected to all of the other steps.

[Definition: A *connection* associates an input or output port with some data source.] Such a connection represents a binding between the port's name and the data source as described by various locations, inline expressions, or readable ports.

An input port can be connected to:

- The output port of some other step.
- A fixed, inline document.
- A document read from a URI.
- One of the inputs declared on one of its *ancestors* or a special port provided by an ancestor compound step, for example, "`current`" in a `p:for-each` or `p:viewport`.

When an input accepts a sequence of documents, the documents can come from any combination of these locations.

In contrast, output ports are connected when they are referenced by another input port, *declared output* or other expression and may be connected to:

- The input port or input context of some other step.
- An option assigned with `p:with-option` or a `p:variable` in a compound step.

- A *value template* in an immediately following step. This can be an AVT in an option shortcut, an AVT on a `p:document` element, or a value template in a `p:inline`.
- One of the outputs declared on its container.

As with an input, the output can be a sequence of documents constructed from any combination of the above.

Within the context of a *compound step*, the *declared outputs* of the compound step must describe their connections. The set of possibilities for this connection is exactly the same set as for any other input port within the current *environment*.

6.1. Connections and the Default Readable Port

The *default readable port* is a convenience for pipeline authors. In the document which describes a pipeline, steps are sequential elements and it is very common for the output of one step to form the natural input to the step described by its immediately following sibling. Consider the following fragment:

```
<p:add-xml-base/>

<p:add-attribute attribute-name="element-count"
                 attribute-value="{count(/*/*)}"/>
```

The output from the add XML base step is the natural input to the add-attribute step. The add XML base step is the source for both the document that will be processed by the add-attribute step and the document that will be used as the context item for the expression in the `attribute-value` attribute.

The fact that the output of the default readable port is used by the following step establishes a connection between the add XML base step and the add-attribute step.

However, unlike an explicit binding which *always* forms a connection, the default readable port *only* forms a connection if it is used. If the processor determines that the default readable port is not used, then it must forgo the connection and the steps can run in parallel.

Consider the following fragment:

```
<p:add-xml-base/>

<p:add-attribute attribute-name="class"
                 attribute-value="homepage">
  <p:with-input port="source">
    <p:document href="http://example.com/"/>
  </p:with-input>
```

The output of the add XML base step is still the default readable port, but it isn't the source for the add-attribute step nor is the context item used in evaluating the `attribute-value` option (or any other option, including the default values of unspecified options), so the processor must omit the connection. This leads to increased parallelism and possibly improved performance.

However, it can cause unexpected results when steps have side-effects. Consider this slightly contrived pipeline fragment:

```
<p:file-touch href="tempdoc.stamp"/>

<p:file-copy href="tempdoc.stamp" target="time.stamp"/>

<p:file-delete href="tempdoc.stamp"/>
```

Each of the file steps has a primary output port and consequently provides a *default readable port* to the following step. Even though the file steps don't have input ports, the document on the default readable port is the context item for evaluating the options on each step.

However, an implementation will observe that none of the options use the context item (and there are no inputs). Consequently, there are no connections between these steps and they can be run in an arbitrary order, or even in parallel. Running delete, followed by copy, followed by touch would be perfectly correct but would not have the side-effects expected by the pipeline author.

There's a trade-off here between giving implementations the freedom to execute pipelines more efficiently and not violating user expectations. In practice, this problem only arises when scheduling steps that have side effects, steps with side effects are (relatively) uncommon, and by their nature are impossible for the processor to schedule with complete confidence.

The pipeline author can make the dependencies explicit in the pipeline, which will ensure that the processor schedules the steps in an order that has the desired the side-effects:

```
<p:file-touch name="touchstamp" href="tempdoc.stamp"/>

<p:file-copy name="copystamp" depends="touchstamp"
             href="tempdoc.stamp" target="time.stamp"/>

<p:file-delete depends="copystamp"
               href="tempdoc.stamp"/>
```

Explicitly marking the dependencies between steps that have side effects is good practice.

6.2. Namespace Fixup on XML Outputs

XProc processors are expected, and sometimes required, to perform *namespace fixup* on XML outputs. Unless the semantics of a step explicitly says otherwise:

- The in-scope namespaces associated with a node (even those that are inherited from namespace bindings that appear among its ancestors in the document in which it appears initially) are assumed to travel with that node.
- Changes to one part of a tree (wrapping or unwrapping a node or renaming an element, for example) do not change the in-scope namespaces associated with the descendants of the node so changed.

As a result, some steps can produce XML documents which have no direct serialization (because they include nodes with conflicting or missing namespace declarations, for example). [Definition: To produce a serializable *XML* document, the XProc processor must sometimes add additional namespace nodes, perhaps even renaming prefixes, to satisfy the constraints of *Namespaces in XML*. This process is referred to as *namespace fixup*.]

Implementors are encouraged to perform *namespace fixup* before passing documents between steps, but they are not required to do so. Conversely, an implementation which *does* serialize between steps and therefore must perform such fixups, or reject documents that cannot be serialized, is also conformant.

Except where the semantics of a step explicitly require changes, processors are required to preserve the information in the documents and fragments they manipulate. In particular, the information corresponding to the [Infoset] properties `[attributes]`, `[base URI]`, `[children]`, `[local name]`, `[namespace name]`, `[normalized value]`, `[owner]`, and `[parent]` **must** be preserved.

The information corresponding to `[prefix]`, `[in-scope namespaces]`, `[namespace attributes]`, and `[attribute type]` **should** be preserved, with changes to the first three only as required for *namespace fixup*. In particular, processors are encouraged to take account of prefix information in creating new namespace bindings, to minimize negative impact on prefixed names in content.

Except for cases which are specifically called out in [Steps 3.0], the extent to which namespace fixup, and other checks for outputs which cannot be serialized, are performed on intermediate outputs is *implementation-defined*.

Whenever an implementation serializes XML, for example for pipeline outputs, logging, or as part of steps such as `p:store` or `p:http-request`, it is a dynamic error if that serialization can not be done so as to produce a document which is both well-formed and namespace-well-formed, as specified in *XML* and *Namespaces in XML*.

7. Initiating a pipeline

Initiating a pipeline necessarily involves two activities: static analysis and dynamic evaluation. [Definition: *Static analysis* consists of those tasks that can be performed by inspection of the pipeline alone, including the binding of static options, computation of serialization properties and document-properties, evaluation of `use-when` expressions, performing a static analysis of all XPath expressions, and detecting static errors.] [Definition: *Dynamic evaluation* consists of tasks which, in general, cannot be performed out until a source document is available.]

It is a *static error* (`err:XS0107`) in XProc if any XPath expression or the XSLT *selection pattern* in option `match` on `p:viewport` contains a static error (error in expression syntax, references to unknown variables or functions, etc.). Type errors, even if they are determined during static analysis, **must not** be raised statically by the XProc processor.

There may be an *implementation-defined* mechanism for providing default values for static `p:option`s. If such a mechanism exists, the values provided must match the sequence type declared for the option, if such a declaration exists.

7.1. Evaluating expressions during static analysis

Several kinds of expressions are evaluated during static analysis:

1. The `select` expressions on static options.
2. Value templates in the attributes or descendants of `p:input` and `p:output` and map attributes on those descendants.
3. Expressions in `use-when` attributes used for conditional element exclusion.

For the purposes of evaluating a these expressions, the initial context node, position, and size are all undefined. The *in-scope bindings* are limited to the lexically preceding, statically declared options. There are no available collections.

Options declared as the direct children of `p:library` in imported libraries are considered in-scope for the declarations that follow.

The entire expression must be evaluated without reference to the non-static inputs to the pipeline. Expressions can access documents as long as they are available statically.

Consider:

```
<p:declare-step version="3.0"
                xmlns:p="http://www.w3.org/ns/xproc">
  <p:input port="source"/>
  <p:option name="A" static="true"
            select="5"/>
  <p:option name="B" static="true"
            select="$A + count(doc('doc.xml')//*)" />
  <p:variable name="D" select="count(//*)"/>

  …
</p:declare-step>
```

The value of `$A` will be 5, unless a different value is provided before static analysis. The value of `$B` will be the value of `$A` plus the number of elements in `doc.xml` *which must be successfully resolved during static analysis*. Although `$D` can reference the document provided dynamically on the `source` port, neither `$A` nor `$B` may.

> **Note**
>
> There is no guarantee that the document read from `doc.xml` during static analysis will be the same as the document read later during dynamic evaluation. See Section 4.1, "External Documents" for further discussion.

The results of XProc extension functions may differ during static analysis, as described in the description of each function.

Any errors that occur while evaluating expressions during static analysis will be raised statically.

7.2. Dynamic evaluation of the pipeline

Dynamic evaluation of the pipeline occurs when it begins to process documents. The processor evaluates any expressions necessary to provide all of the input documents and options required. The step processes the input documents and produces outputs which flow through the pipeline.

Unless otherwise specified, expressions that appear in attribute values (*attribute value templates*, map and array initializers that are always treated as expressions, etc.) get their context item from the *default readable port*. If there is no default readable port, the context item is undefined.

7.2.1. Environment

[Definition: The *environment* is a context-dependent collection of information available within subpipelines.]

The environment consists of:

1. A set of readable ports. [Definition: The *readable ports* are a set of step name/port name pairs.] Inputs and outputs can only be connected to readable ports.

2. A default readable port. [Definition: The *default readable port*, which may be undefined, is a specific step name/port name pair from the set of readable ports.]

3. A set of in-scope bindings. [Definition: The *in-scope bindings* are a set of name-value pairs, based on *option* and *variable* bindings.]

[Definition: The *empty environment* contains no readable ports, an undefined default readable port, and no in-scope bindings.]

Unless otherwise specified, the environment of a *contained step* is its *inherited environment*. [Definition: The *inherited environment* of a *contained step* is an environment that is the same as the environment of its *container* with the standard modifications.]

The standard modifications made to an inherited environment are:

1. The declared inputs of the container are added to the *readable ports*.

   In other words, contained steps can see the inputs to their container.

2. The union of all the declared outputs of all of the step's sibling steps are added to the *readable ports*.

   In other words, sibling steps can see each other's outputs in addition to the outputs visible to their container.

3. If there is a preceding sibling step element:

   - If that preceding sibling has a *primary output port*, then that output port becomes the *default readable port*.
   - Otherwise, the *default readable port* is undefined.

4. If there *is not* a preceding sibling step element:

   - If the container has a *primary input port*, the *default readable port* is that *primary input port*.

- Otherwise, the default readable port is unchanged.

A step with no parent inherits the *empty environment*.

Variables and options are lexically scoped. The environment of a step also includes the *in-scope bindings* for all of the variables and options "visible" from its lexical position. Variables and options can shadow each other; only the lexically most recent bindings are visible.

7.2.1.1. Initial Environment

When a pipeline is invoked by a processor, an initial environment is constructed. [Definition: An *initial environment* is a *connection* for each of the *readable ports* and a set of option bindings used to construct the initial *in-scope bindings*.] This environment is used in place of the *empty environment* that might have otherwise been provided.

An invoked pipeline's *initial environment* is different from the environment constructed for the sub-pipeline of a declared step. The initial environment is constructed for the initial invocation of the pipeline by the processor outside the application. Steps that are subsequently invoked construct an environment as specified in Section 16.5.1, "Declaring pipelines".

When constructing an *initial environment*, an implementation is free to provide any set of mechanisms to construct connections for the input ports of the invoked step. These mechanisms are not limited to the variety of mechanisms described within this specification. Any extensions are implementation defined.

The set of *in-scope bindings* are constructed from a set of option name/value pairs. Each option value can be a simple string value, a specific data type instance (e.g. xs:dateTime), or a more complex value like a map item. How these values are specified is implementation defined.

7.2.2. XPath in XProc

XProc uses XPath 3.1 as an expression language. XPath expressions are evaluated by the XProc processor in several places: on compound steps, to compute the default

values of options and the values of variables; on atomic steps, to compute the actual values of options.

XPath expressions are also passed to some steps. These expressions are evaluated by the implementations of the individual steps.

This distinction can be seen in the following example:

```
<p:variable name="home" select="'http://example.com/docs'"/>

<p:load name="read-from-home">
  <p:with-option name="href" select="concat($home,'/document.xml')"/>
</p:load>

<p:split-sequence name="select-chapters" test="@role='chapter'">
  <p:with-input port="source" select="//section"/>
</p:split-sequence>
```

The select expression on the variable "`home`" is evaluated by the XProc processor. The value of the variable is "`http://example.com/docs`".

The `href` option of the `p:load` step is evaluated by the XProc processor. The actual `href` option received by the step is simply the string literal "`http://example.com/docs/document.xml`". (The select expression on the `source` input of the `p:split-sequence` step is also evaluated by the XProc processor.)

The XPath expression "`@role='chapter'`" is passed literally to the `test` option on the `p:split-sequence` step. That's because the nature of the `p:split-sequence` is that *it evaluates* the expression. Only some options on some steps expect XPath expressions.

The XProc processor evaluates all of the XPath expressions in `select` attributes on variables, options, and inputs, in `match` attributes on `p:viewport`, and in `test` attributes on `p:when` and `p:if` steps.

See Appendix B, XPath contexts in XProc for a detailed description of the context.

8. XPath Extension Functions

The XProc processor **must** support the additional functions described in this section in XPath expressions evaluated by the processor.

These functions **must not** be supported in XPath expressions evaluated by a step. In the interest of interoperability and to avoid imposing unnecessary constraints on implementors, XPath expressions inside, for example, a template in an XSLT step, cannot be aware of the XProc-defined functions.

8.1. System Properties

XPath expressions within a pipeline document can interrogate the processor for information about the current state of the pipeline. Various aspects of the processor are exposed through the `p:system-property` function:

p:system-property($property as *xs:string*) as *xs:string*

The `$property` string **must** have the form of an EQName. If it is a QName, it is expanded using the namespace declarations in scope for the expression. It is a *dynamic error* (`err:XD0015`) if a QName is specified and it cannot be resolved with the in-scope namespace declarations. The `p:system-property` function returns the string representing the value of the system property identified by the EQName. If there is no such property, the empty string **must** be returned.

Implementations **must** provide the following system properties, which are all in the XProc namespace:

`p:episode`
: Returns a string which **should** be unique for each invocation of the pipeline processor. In other words, if a processor is run several times in succession, or if several processors are running simultaneously, each invocation of each processor should get a distinct value from `p:episode`.

  The unique identifier must be a valid XML name.

`p:locale`
: Returns a string which identifies the current environment (usually the OS) language. This is useful for, for example, message localization purposes. The exact format of the language string is *implementation-defined* but **should** be consistent with the `xml:lang` attribute.

`p:product-name`
: Returns a string containing the name of the implementation, as defined by the implementer. This should normally remain constant from one release of the product to the next. It should also be constant across platforms in cases where the same source code is used to produce compatible products for multiple execution platforms.

`p:product-version`
: Returns a string identifying the version of the implementation, as defined by the implementer. This should normally vary from one release of the product to the next, and at the discretion of the implementer it may also vary across different execution platforms.

`p:vendor`
: Returns a string which identifies the vendor of the processor.

`p:vendor-uri`
: Returns a URI which identifies the vendor of the processor. Often, this is the URI of the vendor's web site.

`p:version`
: Returns the version(s) of XProc implemented by the processor as a space-separated list. For example, a processor that supports XProc 1.0 would return "1.0"; a processor that supports XProc 1.0 and 3.0 would return "1.0 3.0"; a processor that supports only XProc 3.0 would return "3.0".

`p:xpath-version`
: Returns the version(s) of XPath implemented by the processor for evaluating XPath expressions on XProc elements. The result is a space-separated list of versions supported. For example, a processor that only supports XPath 3.1 would return "3.1"; a processor that supports XPath 3.1 and XPath 3.2 could return "3.1 3.2".

`p:psvi-supported`
: Returns true if the implementation supports passing PSVI annotations between steps, false otherwise.

Implementations may support additional system properties but such properties **must** be in a namespace and **must not** be in the XProc namespace.

The `p:system-property` function behaves normally during static analysis. It is *implementation-defined* which additional system properties are available during static analysis. If an additional system property is not available during static analysis, an empty string **must** be returned.

8.2. Step Available

The `p:step-available` function reports whether or not a particular type of step is understood by the processor and in scope where the function is called.

p:step-available($step-type as *xs:string*) as *xs:boolean*

The `$step-type` string **must** have the form of an EQName. If it is a QName, it is expanded using the namespace declarations in scope for the expression. It is a *dynamic error* (`err:XD0015`) if a QName is specified and it cannot be resolved with the in-scope namespace declarations. The `p:step-available` function returns `true` if and only if the processor knows how to evaluate a step of the specified type where the function is called.

In case the argument of `p:step-available` refers to a step that is currently being defined, the function returns `false`. In practice this occurs only if:

- `p:step-available` is used in a `use-when` expression on or within the `p:declare-step` that defines the step to which it refers.
- `p:step-available` is used in a `use-when` expression on a `p:library` that contains the declaration of the step to which it refers.

The `p:step-available` behaves normally during static analysis.

8.3. Iteration Position

Some compound steps, such as `p:for-each` and `p:viewport`, process a sequence of documents. The iteration position is the position of the current document in that sequence: the first document has position 1, the second 2, etc. The `p:iteration-position` function returns the iteration position of the nearest compound step that processes a sequence of documents.

p:iteration-position() as *xs:integer*

If there is no compound step that processes a sequence of documents among the ancestors of the element on which the expression involving `p:iteration-position` occurs, it returns 1.

The value of the `p:iteration-position` function during static analysis is 1.

8.4. Iteration Size

Both `p:for-each` and `p:viewport` process a sequence of documents. The iteration size is the total number of documents in that sequence. The `p:iteration-size` function returns the iteration size of the nearest ancestor compound step that processes a sequence of documents.

p:iteration-size() as *xs:integer*

If there is no `p:for-each` or `p:viewport` among the ancestors of the element on which the expression involving `p:iteration-size` occurs, it returns 1.

The value of the `p:iteration-size` function during static analysis is 1.

8.5. Version Available

Returns true if and only if the processor supports the XProc version specified.

p:version-available($version as *xs:string*) as *xs:boolean*

A version 3.0 processor will return `true()` when `p:version-available('3.0')` is evaluated.

The `p:version-available` function behaves normally during static analysis.

8.6. XPath Version Available

Returns true if and only if the processor supports the XPath version specified.

p:xpath-version-available($version as *xs:string*) as *xs:boolean*

A processor that supports XPath 3.1 will return `true()` when `p:xpath-version-available('3.1')` is evaluated.

The `p:xpath-version-available` function behaves normally during static analysis.

8.7. Document properties

This function retrieves the *document properties* of a document as a map.

p:document-properties($doc as *item()*) as *map(xs:QName,item()*)*

The map returned contains (exclusively) the document properties associated with the `$doc` specified. If the item is not associated with a document, the resulting map will be empty.

Document properties are associated with documents that flow out of steps. Documents loaded with XPath functions or through other out-of-band means may not have properties associated with them. In order to provide a consistent interface for pipeline authors, the base URI of a node is always returned in the `base-uri` property and the `content-type` property always contains at least the most general appropriate content type: If the document node has a single text node child, `text/plain` is used, `application/xml` otherwise.

The `p:document-properties` function behaves normally during static analysis.

8.8. Document property

This function retrieves a single value from the *document properties* of a document.

p:document-property($doc as *item()*, $key as *item()*) as *item()**

The item returned is the value of the property named `$key` in the document properties. An empty sequence is returned if `$doc` is not associated with a document or no such key exists. `$key` is interpreted as follows:

- If `$key` is of type `xs:QName`, its value is used unchanged.
- If `$key` is an instance of type `xs:string` (or a type derived from `xs:string`) its value is transformed into a `xs:QName` using the XPath EQName production rules. That is, it can be written as a local-name only, as a prefix plus local-name or as a URI plus local-name (using the `Q{}` syntax).

  It is a *dynamic error* (`err:XD0061`) if `$key` is of type `xs:string` and cannot be converted into a `xs:QName`.
- If `$key` is of any other type, the function returns the empty sequence.

The `p:document-property` function behaves normally during static analysis.

8.9. Transform file system paths into URIs and normalize URIs

Most web technologies identify resources with URIs, but XProc must also operate with resources that are identified with strings encoded in other ways, for example, file system paths and the names of resources in archive files.

The `p:urify` function attempts to transform file system paths into file URIs ([RFC 3986]). If a presumptive yet not fully compliant URI is given as an argument, `p:urify` attempts to resolve the string into a URI.

The `p:urify` function resolves a string into a URI by employing a series of heuristics. These have been selected so that `p:urify` will not corrupt any actual, valid URIs and with the goal that it will return the least surprising result for any other string. If a pipeline author has more context to determine how a string should be transformed

into a URI, writing the conversion process “by hand” in the pipeline may achieve better results.

p:urify($filepath as *xs:string*, $basedir as *xs:string?*) as *xs:string*

p:urify($filepath as *xs:string*) as *xs:string*

If the single-argument version of the function is used, the result is the same as calling the two-argument version with `$basedir` set to the empty sequence.

The `p:urify` function behaves normally during static analysis.

The heuristics that `p:urify` performs occur in two stages: first, the input string is analyzed to identify its features, then these features are used to construct a final URI string. An additional “fixup” process may be performed on the path portion of the URI.

To make the operation of the `p:urify` function easier to understand, the description that follows is presented as an algorithm with regular expressions. Implementors are not required to implement it this way, any implementation that achieves the correct results can be used.

The function may be implemented as an operation on strings; it need not try to determine the existence of a file or directory, and it **should not** follow symbolic links. However, two pieces of information need to be known from the environment: Whether the operating system identifies as “Windows” and the value of the file separator. More precisely, the operating system identifies as Windows if the `os-name` property as returned by the `p:os-info` steps starts with the string “`Windows`”. The file separator is what `p:os-info` returns as the `file-separator` property. If either of them are not known, it is assumed that the operating system is not Windows and the file separator is the forward slash, “`/`”.

The comparisons and regular expressions that follow are presented in lower case, but all of the operations performed are case blind: “file”, “FILE”, “File”, and “FiLe” are all identical.

8.9.1. Normalize file separators

Before beginning analysis, if the system file separator is not "/", then all occurrences of the file separator in filenames must be replaced by "/". (If the file separator *is* "/", this section does not apply.)

Replacing file separators with "/" simplifies the analysis that follows and assures that the resulting URI will be syntactically correct. However, it must only be done when it is determined that the `$filepath` will become part of a file: URI.

To determine if the path will become part of a file: URI, consider the following cases in turn, stopping at the first which applies:

- If the `$filepath` begins with "file:" or, on Windows, if it begins with a single letter followed by a colon, it will be part of a file: URI.
- If the `$filepath` begins with an explicit scheme other than "file", it will not be part of a file: URI.
- If the `$basedir` is absent or the empty string, it will be part of a file: URI.
- If the `$basedir` begins with an explicit scheme other than "file", it will not be part of a file: URI.
- If none of the preceding cases applies, it will be part of a file: URI.

If it has been determined that the path will become part of a file: URI, replace each occurrence of the file separator character in `$filepath` with a "/".

8.9.2. Analysis

The `$filepath` presented is analyzed to identify the following features:

- The *scheme*, which may be absent or implicitly known or explicitly known.
- Whether or not the string can be interpreted as *hierarchical*.
- The *authority*, which may be absent.
- The *drive letter*, which may be absent.

- And the *path*, which may be *absolute* or *relative*.

The analysis proceeds along the following lines, stopping as soon as the features have been identified.

1. If the `$filepath` is the empty string, it has no scheme, no authority, and no drive letter. Its path is the empty string and it is relative and hierarchical.

2. If the `$filepath` is the string "`//`", it has no scheme, no authority, and no drive letter. Its path is the string "`/`" and it is absolute and hierarchical.

3. On a Windows system, if the `$filepath` matches the regular expression "`^(file:/*)?([a-z]):(.*)`", it is a "file" scheme URI. If the first match group begins with "file:", it is an explicit file scheme URI, otherwise it is an implicit file scheme URI. The drive letter is the second match group. If the third match group begins with a "`/`", the path is absolute and consists of the third match group with all but one leading "`/`" removed. Otherwise, the path is relative and consists of the entire third match group, or the empty string if the third match group is empty.

   In all cases, the scheme is hierarchical and the authority is absent.

   For example:

   - `C:Users/Jane/Documents and Files/Thing`, an implicit file URI on drive C with the relative path "Users/Jane/Documents and Files/Thing".
   - `C:/Users/Jane/Documents and Files/Thing` an implicit file URI on drive C with the absolute path "/Users/Jane/Documents and Files/Thing".
   - `C://Users/Jane/Documents and Files/Thing` an implicit file URI on drive C with the absolute path "/Users/Jane/Documents and Files/Thing".
   - `C:///Users/Jane/Documents and Files/Thing` an implicit file URI on drive C with the absolute path "/Users/Jane/Documents and Files/Thing".
   - `file:C:Users/Jane/Documents and Files/Thing` an explicit file URI on drive C with the relative path "Users/Jane/Documents and Files/Thing".

- `file:C:/Users/Jane/Documents and Files/Thing` an explicit file URI on drive C with the absolute path "/Users/Jane/Documents and Files/Thing".
- `file:C://Users/Jane/Documents and Files/Thing` an explicit file URI on drive C with the absolute path "/Users/Jane/Documents and Files/Thing".
- `file:C:///Users/Jane/Documents and Files/Thing` an explicit file URI on drive C with the absolute path "/Users/Jane/Documents and Files/Thing".

4. If the *`$filepath`* matches the regular expression "`^file://([^/]+)(/.*)?$`", it is an explicit "file" scheme URI. The first match group is the authority. If the second match group begins with a "/", the path is absolute and consists of the second match group with all but one leading "/" removed. Otherwise, the path is the empty string and is relative.

   In all cases, the scheme is hierarchical and the drive letter is absent.

   For example:

   - `file://authority.com` an explicit file URI with the authority "authority.com" and the relative path "".
   - `file://authority.com/` an explicit file URI with the authority "authority.com" and the absolute path "/".
   - `file://authority.com/path/to/thing` an explicit file URI with the authority "authority.com" and the absolute path "/path/to/thing".
   - `file://authority.com//path/to/thing` an explicit file URI with the authority "authority.com" and the absolute path "/path/to/thing".
   - `file://authority.com///path/to/thing` an explicit file URI with the authority "authority.com" and the absolute path "/path/to/thing".
   - `file://authority.com:8080/path/to/thing` an explicit file URI with the authority "authority.com:8080" and the absolute path "/path/to/thing".

5. If the *`$filepath`* matches the regular expression "`^file:(.*)$`", it is an explicit "file" scheme URI. If the first match group begins with a "/", the path is

absolute and consists of the first match group with all but one leading "/" removed. Otherwise, the path is relative and consists of the entire first match group, or the empty string if the first match group is empty.

In all cases, the scheme is hierarchical and the drive letter and authority are absent.

For example:

- `file:` is an explicit file URI with the relative path "".
- `file:path/to/thing` is an explicit file URI with the relative path "path/to/thing".
- `file:/path/to/thing` is an explicit file URI with the absolute path "/path/to/thing".
- `file://path/to/thing` does not apply. It matches the preceding case; "path" is the authority.
- `file:///path/to/thing` is an explicit file URI with the absolute path "/path/to/thing".

6. If the *`$filepath`* matches the regular expression "`^([a-z]+):(.*)$`", it is an explicit URI in the scheme identified by the first match group. The path is the second match group. (In the terms of [RFC 3986], it may have an authority component, but that's not relevant to `p:urify`.) If the implementation does not know if the scheme is hierarchical, it is considered hierarchical if the path contains a "/", otherwise it is considered non-hierarchical. (The "http", "https", and "ftp" schemes are hierarchical, for example; the "mailto", "urn" and "doi" schemes are not.)

   In all cases, the drive letter and authority are absent.

   For example:

   - `urn:publicid:ISO+8879%3A1986:ENTITIES+Added+Latin+1:EN` is a non-hierarchical "urn" URI. By the heuristic applied, the path is "publicid:ISO+8879%3A1986:ENTITIES+Added+Latin+1:EN", but this will

never be relevant as relative and absolute path resolution is never applied to non-hierarchical schemes.

- `https:` is a hierarchical "https" URI with the relative path "".
- `https://example.com` is a hierarchical "https" URI with the absolute path "//example.com".
- `https://example.com/` is a hierarchical "https" URI with the absolute path "//example.com/".
- `https://example.com/path/to/thing` is a hierarchical "https" URI with the absolute path "//example.com/path/to/thing".
- `https://example.com//path/to/thing` is a hierarchical "https" URI with the absolute path "//example.com//path/to/thing".
- `https://example.com:9000/path/to/thing` is a hierarchical "https" URI with the absolute path "//example.com:9000/path/to/thing".

7. If the *`$filepath`* matches the regular expression "`^//([^/]+)(/.*)?$`", it has no scheme. The first match group is the authority. If the second match group begins with a "/", the path is absolute and consists of the second match group with all but one leading "/" removed. Otherwise, the path is the empty string and is relative. It has no drive letter.

   In all cases, the URI is hierarchical and the drive letter is absent.

   For example:

   - `//authority` has the authority "authority" and the relative path "".
   - `//authority/` has the authority "authority" and the absolute path "/".
   - `//authority/path/to/thing` has the authority "authority" and the absolute path "/path/to/thing".
   - `//authority//path/to/thing` has the authority "authority" and the absolute path "/path/to/thing".

- `//authority///path/to/thing` has the authority "authority" and the absolute path "/path/to/thing".
- `//authority:8080/path/to/thing` has the authority "authority:8080" and the absolute path "/path/to/thing".
- `//authority/Documents and Files/thing` has the authority "authority" and the absolute path "/Documents and Files/thing".

8. If the *`$filepath`* begins with a "/", the path is absolute and consists of *`$filepath`* with all but one leading "/" removed. Otherwise, the path is relative and consists of the entire *`$filepath`*. It is hierarchical and has no scheme, no authority and no drive letter. (This condition always applies if no preceding condition does.)

   For example:

   - `path/to/thing` has the relative path "path/to/thing".
   - `/path/to/thing` has the absolute path "/path/to/thing".
   - `//path/to/thing` does not apply. It matches the preceding case; "path" is the authority.
   - `///path/to/thing` has the absolute path "/path/to/thing".
   - `Documents and Files/thing` has the relative path "/Documents and Files/thing".
   - `/Documents and Files/thing` has the absolute path "/Documents and Files/thing".

If the analysis determines that the string represents a non-hierarchical URI, the *`$filepath`* is returned unchanged.

If the analysis determines that the scheme is known and the path is absolute, a URI is constructed from the features, see below. Otherwise, the URI must be made absolute with respect to the *`$basedir`* provided.

If the *`$basedir`* is the empty sequence, construct a presumptive URI string from the string that represents the current working directory. If this presumptive URI does not

end with the file separator, append the file separator. If the implementation is running in an environment where the concept of "current working directory" does not apply, the presumptive URI is the empty string. This presumptive URI becomes the `$basedir`.

Analyze the features of the `$basedir`.

If the `$basedir` has no scheme, it's implicitly a "file" URI.

If the `$basedir` path is relative, the `$filepath` cannot be made absolute. It is a *dynamic error* (`err:XD0074`) if no absolute base URI is supplied to `p:urify` and none can be inferred from the current working directory.

The following additional constraints apply.

- It is a *dynamic error* (`err:XD0075`) if the relative path has a drive letter and the base URI has a different drive letter or does not have a drive letter.
- It is a *dynamic error* (`err:XD0076`) if the relative path has a drive letter and the base URI has an authority or if the relative path has an authority and the base URI has a drive letter.
- It is a *dynamic error* (`err:XD0077`) if the relative path has a scheme that differs from the scheme of the base URI.
- It is a *dynamic error* (`err:XD0080`) if the `$basedir` has a non-hierarchical scheme.

Combine the features of the `$filepath` with the features of the `$basedir` to obtain a set of features to use to construct the result.

If the `$filepath` has no scheme or an implicit file scheme, perform fixup on the path, as described below. If the `$basedir` is an implicit file URI, perform fixup on its path.

1. The scheme is the scheme of the `$basedir`.
2. If the `$filepath` has an authority, use that authority, otherwise use the authority of the `$basedir`, if it has one.
3. The drive letter is the drive letter of the `$basedir`.

4. If the path of the `$filepath` absolute, that's the path. Otherwise the path is the path of the `$filepath` resolved against the `$basedir`.

   If the `$basedir` ends in "/", the resolved path is the concatention of the `$basedir` and `$filepath`'s path. Otherwise, the resolved path is the concatentation of all the characters in `$basedir` up to and including the last "/" it contains with the `$filepath`'s path.

   Path contraction for "." and ".." is performed on the resolved path according to Section 3.3 of [RFC 3986].

8.9.3. Path fixup

If fixup is performed, the characters "?", "#", "\" and " " (space) are replaced by their percent-encoded forms, "%3F", "%23", "%5C", and "%20", respectively.

Unreserved characters that are percent encoded in the path are decoded per Section 2.4 of [RFC 3986].

8.9.4. URI construction

The `p:urify` result string is constructed from the features of the path (or the features of the path as resolved against the `$basedir`, if applicable) in the following way:

1. Begin with an empty string.
2. If there is a scheme, append the scheme followed by a ":".
3. If there is an authority, append "//" followed by the authority.
   - On a Windows system, if the scheme is known to be `file` and the processor determines that the authority component is accessible via the universal naming convention (UNC), an additional "//" **may** be added before the authority. (In other words, `file:////uncserver` is allowed.)
4. If there is *not* an authority, but the scheme is "file" and the path is absolute,
   - Append "//".

- If there is a drive letter, append another "/".

5. If there is a drive letter, append the drive letter followed by a ":".
6. Append the path.

The string constructed is the `p:urify` result.

8.10. Function library importable

The `p:function-library-importable` function reports whether or not function libraries of a particular type can be imported.

p:function-library-importable($library-type as *xs:string*) as *xs:boolean*

The `$library-type` string is interpreted as a content type. If the processor understands (*i.e.* if `p:import-functions` understands) how to load function libraries of that type, this function returns `true()`, otherwise it returns `false()`.

The `p:function-library-importable` function behaves normally during static analysis.

8.11. Other XPath Extension Functions

It is *implementation-defined* if the processor supports any other XPath extension functions. Additional extension functions, if any, **must not** use any of the XProc namespaces.

The value of the any other XPath extension functions during static analysis is *implementation-defined*.

9. PSVIs in XProc

XML documents flow between steps in an XProc pipeline. Section A.3, "Infoset Conformance" identifies the properties of those documents that **must** be available. Implementations **may** also have the ability to pass PSVI annotations between steps.

Whether or not the pipeline processor supports passing PSVI annotations between steps is *implementation-defined*. The exact PSVI properties that are preserved when documents are passed between steps is *implementation-defined*.

A pipeline can use the `p:psvi-supported` system property to determine whether or not PSVI properties can be passed between steps.

A pipeline can assert that PSVI support is required with the `psvi-required` attribute:

- On a `p:declare-step`, `psvi-required` indicates whether or not the declared step requires PSVI support. It is a *dynamic error* (`err:XD0022`) if a processor that does not support PSVI annotations attempts to invoke a step which asserts that they are required.
- On a `p:library`, the `psvi-required` attribute provides a default value for all of its `p:declare-step` *children* that do not specify a value themselves.

Many of the steps that an XProc pipeline can use are transformative in nature. The `p:delete` step, for example, can remove elements and attributes; the `p:label-elements` step can add attributes; etc. If PSVI annotations were always preserved, the use of such steps could result in documents that were inconsistent with their schema annotations.

In order to avoid these inconsistencies, most steps **must not** produce PSVI annotated results even when PSVI passing is supported.

If PSVI passing is supported, the following constraints apply:

1. Implementations **must** faithfully transmit any PSVI properties produced on step outputs to the steps to which they are connected.
2. When only a subset of the input is processed by a step (because a `select` expression appears on an input port or a `match` expression is used to process only part of the input), any PSVI annotations that appear on the selected input **must** be preserved in the resulting documents passed to the step.

   Note that ID/IDREF constraints, and any other whole-document constraints, may not be satisfied within the selected portion, irrespective of what its PSVI properties claim.

3. If an output of a compound step is connected to an output which includes PSVI properties, those properties **must** be preserved on the output of the compound step, *except* for the output of `p:viewport` which **must not** contain any PSVI properties.

4. If an implementation supports XPath 2.0 or later, the data model constructed with which to evaluate XPath expressions and *selection patterns* **should** take advantage of as much PSVI information as possible.

   [Definition: A *selection pattern* uses a subset of the syntax for path expressions, and is defined to match a node if the corresponding path expression would select the node. It is defined as in the XSLT 3.0 specification.]

5. All steps that explicitly behave like the `p:identity` step under some circumstances must preserve PSVI properties under those circumstances. In this specifications, those steps are `p:if` when the condition is false and `p:choose` when no subpipline is selected.

6. Except as specified above, or in the descriptions of individual steps, implementations **must not** include PSVI properties in the outputs of steps. It is *implementation-defined* what PSVI properties, if any, are produced by extension steps.

> **Note**
>
> A processor that supports passing PSVI properties between steps is always free to do so. Even if `psvi-required="false"` is explicitly specified, it is not an error for a step to produce a result that includes additional PSVI properties, provide it does not violate the constraints above.

10. Value Templates

An attribute or text node in a pipeline may, in particular circumstances, contain embedded expressions enclosed between curly brackets. Attributes and text nodes

that use (or are permitted to use) this mechanism are referred to respectively as *attribute value templates* (AVTs) and *text value templates.* (TVTs).

[Definition: Collectively, attribute value templates and text value templates are referred to as *value templates*.]

A value template is a string that contains zero or more expressions delimited by curly brackets. Outside an expression, a doubled left or right curly bracket ("{{" or "}}") represents a literal, single bracket and does not start or end an expression. Once an expression begins, it extends to the first unmatched right curly bracket that is not within a string literal or comment.

Value templates are not recursive. Curly brackets inside an expression are part of that expression and are not recognized as nested value templates.

It is a *static error* (`err:XS0066`) if an expression does not have a closing right curly bracket or if an unescaped right curly bracket occurs outside of an expression.

It is a static error if the string contained between matching curly brackets in a value template, when interpreted as an XPath expression, contains errors. The error is signaled using the appropriate XPath error code.

It is a *dynamic error* (`err:XD0050`) if the XPath expression in a value template can not be evaluated.

It is a *dynamic error* (`err:XD0051`) if the XPath expression in an AVT or TVT evaluates to something to other than a sequence containing atomic values or nodes. Function, array and map items are explicitly excluded here because they do not have a string representation.

A value template that contains a reference to the context item reads that context item from the *default readable port*. This establishes a connection between the two steps. It is a *dynamic error* (`err:XD0065`) to refer to the context item, size, or position in a value template if a sequence of documents appears on the default readable port. Value templates that do not contain a reference to the context item do not establish a connection to another step and do not require that a *default readable port* is available.

10.1. Attribute Value Templates

[Definition: In an attribute that is designated as an *attribute value template*, an expression can be used by surrounding the expression with curly brackets (`{}`), following the general rules for *value templates*].

Curly brackets are not treated specially in an attribute value in an XProc pipeline unless the attribute is specifically designated as one that permits an attribute value template. Option shortcuts permit attribute value templates. Whether or not an extension attribute permits attribute value templates is *implementation-defined*. In element syntax summaries in this specification, the value of an attribute that allows attribute value templates is surrounded by curly brackets.

An attribute value template can be seen as an alternating sequence of zero or more "fixed" (non-expression) parts and expression parts.

The result of the attribute value template is the concatenation of the fixed parts and the string-value of the result of evaluating each expression part.

> **Note**
>
> This process can generate dynamic errors, for example if the sequence contains an element with a complex content type (which cannot be atomized).

The value of an attribute that contains attribute value templates is a single string (the concatenation of the string values of the evaluated templates and non-template parts) as an `xs:untypedAtomic`.

10.2. Text Value Templates

[Definition: In a text node that is designated as a *text value template*, expressions can be used by surrounding each expression with curly brackets (`{}`), following the general rules for *value templates*.]

Text nodes that are descendants of a `p:inline` and text nodes that are descendants of an element node in an implicit inline may be text value templates. No other text node is a text value template.

Whether or not a text node that may be a text value template is designated one is determined by `expand-text` and `p:inline-expand-text` attributes, see Section 14.9.1, "Expand text attributes".

A text value template can be seen as an alternating sequence of zero or more "fixed" (non-expression) parts and expression parts.

This produces a sequence of strings (the fixed parts) and items (the results of evaluating each expression). Any items that are non-string atomic values are converted to strings by taking their string value. Strings are converted into text nodes.

The result of the text value template is this sequence of nodes.

> **Note**
>
> Unlike XSLT, in XProc, text value templates are not atomized and converted to single text nodes. It is possible to insert nodes with text value templates in XProc, for example, if the XPath expressions refer to variables that have node content.

If a node to be inserted with a text value template is a document node, all the children of the document node are inserted.

How the nodes are inserted depends on the content type of the `p:inline`.

1. If the content type is an *XML media type* or an *HTML media type*, the nodes are added to the XML document where they occur. This is analogous to the way element constructors work in [XQuery 1.0].

   If the node is an attribute it is added to an element parent if and only if the attribute either has no preceding nodes in the sequence of nodes or has only attributes as preceding nodes. It is a *dynamic error* (`err:XD0052`) if the XPath

expression in a TVT evaluates to an attribute and either the parent is not an element or the attribute has a preceding node that it not an attribute.

2. If the content type is not an *XML media type* or an *HTML media type*, each text value template is replaced by the concatenation of the serialization of the nodes that result from evaluating the template.

   This serialization is performed with the following serialization parameters:

Parameter	Value
`byte-order-mark`	false
`cdata-section-elements`	()
`doctype-public`	()
`doctype-system`	()
`encoding`	“utf-8”
`escape-uri-attributes`	false
`include-content-type`	false
`indent`	false
`media-type`	“application/xml”
`method`	“xml”
`normalization-form`	()
`omit-xml-declaration`	true
`standalone`	false
`undeclare-prefixes`	false
`use-character-maps`	()
`version`	1.0

Interpretation of the character content of the `p:inline` according to the media type occurs after text value templates have been replaced.

Examples

Consider the following examples. In each case:

- The variable `$name` is bound to the following XML element:

```
<name><given>Mary</given> <surname>Smith</surname></name>
```

- The result of evaluating the text value template "`{$name/node()}`" is a sequence of three nodes, the given name element, a text node containing a single space, and the surname element.

If the media type is an XML media type:

```
<p:inline content-type="application/xml">
  <attribution>{$name/node()}</attribution>
</p:inline>
```

the result is that sequence of nodes:

```
<attribution><given>Mary</given> <surname>Smith</surname></attribution>
```

If the media type is not an XML media type:

```
<p:inline content-type="application/json">
  {{ "name": "{$name/node()}" }}
</p:inline>
```

the result is the concatenation of the serialization of the nodes:

```
{ "name": "<given>Mary</given> <surname>Smith</surname>" }
```

If the string value is desired, instead of escaped markup, write the expression such that it returns the string values:

```
<p:inline content-type="application/json">
  {{ "name": "{$name/node()/string()}" }}
</p:inline>
```

To produce:

```
  { "name": "Mary Smith" }
```

11. Variables and Options

11.1. Variables

Pipeline authors can create variables to hold computed values.

[Definition: A *variable* is a name/value pair. The name **must** be an expanded name. The value may be any XPath data model value.] Variable names are always expressed as literal values, pipelines cannot construct variable names dynamically.

The names of variables and options are not distinct and are lexically scoped. [Definition: We say that a variable *shadows* another variable (or option) if it has the same name and appears later in the same lexical scope.]

Consider this pipeline:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" version="3.0">

  <p:option name="bname" as="xs:integer" select="1"/>
  <p:identity message="NAME1={$bname}">
    <p:with-input port="source">
      <p:empty/>
    </p:with-input>
  </p:identity>


  <p:variable name="bname" select="$bname + 1"/>
  <p:identity message="NAME2={$bname}"/>

  <p:variable name="bname" select="7"/>
  <p:identity message="NAME3={$bname}"/>

</p:declare-step>
```

If no overriding value is provided for `$bname` at runtime, the pipeline will produce three messages: “NAME1=1”, “NAME2=2”, and “NAME3=7”. (If an overriding

value is provided at runtime, “NAME1” will have that value, “NAME2” will have one more than that value, and “NAME3” will have the value 7.

11.2. Options

Some steps accept options. The value of an option is the default value specified in its declaration, or a value provided by the caller of the step (overriding the default). If it has neither a default value nor a provided value, its value is the empty sequence.

[Definition: An *option* is a name/value pair. The name **must** be an expanded name. The value may be any XPath data model value.] Option names are always expressed as literal values, pipelines cannot construct option names dynamically.

How outside values are specified for pipeline options on the pipeline initially invoked by the processor is *implementation-defined*. In other words, the command line options, APIs, or other mechanisms available to specify such options values are outside the scope of this specification.

Some steps require a set of name/value pairs for the operations they perform. For example, an XSLT stylesheet might have required parameters or an XQuery query might have external variables. In the XProc Step Library, the standard way to pass such values to the step is to use an option named “`parameters`” whose value is a map.

11.3. Static Options

A `p:option` may be declared “static”; options declared within a `p:library` **must** be static.

It is a *static error* (`err:XS0109`) if options that are the direct children of `p:library` are not declared “static”.

The values of static options are computed during static analysis.

XProc defines a single, global scope for static options. Every static option must have exactly one in-scope declaration.

11.4. Variable and option types

Variables and options may declare that they have a type using the `as` attribute. The attribute value **must** be an [XPath 3.1] sequence type. It is a *static error* (`err:XS0096`) if the sequence type is not syntactically valid. The sequence type `item()*` is assumed if no explicit type is provided.

If a variable or option declares a type, the supplied value of the variable or option is converted to the required type, using the function conversion rules specified by XPath 3.1. It is a *dynamic error* (`err:XD0036`) if the supplied or defaulted value of a variable or option cannot be converted to the required type.

11.5. Implicit casting

For the most part, the rules for casting between types in XPath work the way authors expect. You can type "3" for a decimal, you don't have to type "3.0", and an untyped atomic value, for example in an attribute such as `limit="3"`, is implicitly cast to a number if that's the required type. You don't have to type `limit="{xs:integer(3)}"`.

Unfortunately, there are two common cases in XProc that are not handled by the XPath conversion rules: conversions from strings to QNames or URIs. (Technically, conversions from `xs:untypedAtomic` or `xs:string`, or from types derived from `xs:string`, values to `xs:QName` or `xs:anyURI` values.) XProc defines additional implicit casting rules for the case where an expression is evaluated to provide the value of a variable or option. (These are not extensions to the XPath rules in the general case and do not apply in arbitrary expressions.)

11.5.1. Special rules for casting QNames

Some steps have options whose values are QNames, for example "`attribute-name`" on `p:add-attribute`. If the type `xs:QName` was strictly enforced, they would be tedious to specify. As a convenience for pipeline authors, the values of variables or options declared with the type `xs:QName` are processed specially. The type `xs:QName` is treated as `xs:anyAtomicType` for the purpose of atomization. The value (or values) are converted to `xs:QNames`:

1. If the value supplied for the option is an instance of `xs:QName` then that value is used.

2. If the value supplied for the option is an instance of `xs:untypedAtomic` or `xs:string` (or a type derived from `xs:string`), the QName is constructed by following the EQName production rules in [XPath 3.1]. That is, it can be written as a local-name only, as a prefix plus local-name, or as a URI qualified name (using the `Q{namespace}local-name` syntax). If it is written as local-name only, the constructed QName will not have a namespace URI, i.e. the default namespace is not applied here. It is a *dynamic error* (`err:XD0061`) if the string value is not syntactically an EQName. It is a *dynamic error* (`err:XD0069`) if the string value contains a colon and the designated prefix is not declared in the in-scope namespaces.

3. It is a *dynamic error* (`err:XD0068`) if the supplied value is not an instance of `xs:QName`, `xs:anyAtomicType`, `xs:string` or a type derived from `xs:string`.

As an additional convenience, if the specified sequence type of an option or a variable is a map with `xs:QName` keys (`map(xs:QName, …)`), the supplied map value is processed specially. This makes it possible to pass in maps using (easier to write) `xs:string` type keys that are converted automatically into the required `xs:QName` keys.

Every key/value pair in a map supplied to a variable or an option with sequence type `map(xs:QName, …)` is processed as follows:

- If the entry's key is of type `xs:QName`, the entry is left unchanged.

- If the entry's key is an instance of type `xs:untypedAtomic` or `xs:string` (or a type derived from `xs:string`) it is transformed into an `xs:QName` using the XPath EQName production rules as described above.

- If the entry's key is of any other type, the entry is ignored and will be removed from the map.

11.5.2. Special rules for casting URIs

Many steps have options whose values are `xs:anyURI` values, for example "`href`" on `p:http-request`. If the type `xs:anyURI` was strictly enforced, they would be tedious to specify. As a convenience for pipeline authors, the values of variables or options declared with the type `xs:anyURI` are processed specially. The value (or values) are converted to `xs:anyURI`s:

1. If the value supplied for the option is an instance of `xs:anyURI` then that value is used.

2. If the value supplied for the option is an instance of `xs:untypedAtomic` or `xs:string` (or a type derived from `xs:string`), the `xs:anyURI` is constructed by casting the value to an `xs:anyURI`.

The XPath rules for casting string values to URIs apply: The extent to which an implementation validates the lexical form of the `xs:anyURI` is *implementation-defined*.

11.6. Namespaces on variables and options

Variable and option values carry with them not only their literal or computed value but also a set of namespaces. To see why this is necessary, consider the following step:

```
<p:delete xmlns:p="http://www.w3.org/ns/xproc">
  <p:with-option name="match" select="'html:div'"
                 xmlns:html="http://www.w3.org/1999/xhtml"/>
</p:delete>
```

The `p:delete` step will delete elements that match the expression "`html:div`", but that expression can only be correctly interpreted if there's a namespace binding for the prefix "`html`" so that binding has to travel with the option.

The default namespace bindings associated with a variable or option value are computed as follows:

1. If the `select` attribute was used to specify the value and it consisted of a single `VariableReference` (per [XPath 3.1]), then the namespace bindings from the referenced option or variable are used.

2. If the `select` attribute was used to specify the value and it evaluated to a node-set, then the in-scope namespaces from the first node in the selected node-set (or, if it's not an element, its parent) are used.

   The expression is evaluated in the appropriate context, See Section 7.2.2, "XPath in XProc".

3. Otherwise, the in-scope namespaces from the element providing the value are used. (For options specified using syntactic shortcuts, the step element itself is providing the value.)

The default namespace is never included in the namespace bindings for a variable or option value. Unqualified names are always in no-namespace.

Unfortunately, in more complex situations, there may be no single variable or option that can reliably be expected to have the correct set of namespace bindings. Consider this pipeline:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
            xmlns:ex="http://example.org/ns/ex"
            xmlns:h="http://www.w3.org/1999/xhtml"
                type="ex:delete-in-div" version="3.0">
<p:input port="source"/>
<p:output port="result"/>
<p:option name="divchild" required="true"/>

<p:delete>
  <p:with-option name="match" select="concat('h:div/',$divchild)"/>
</p:delete>

</p:declare-step>
```

It defines an atomic step ("`ex:delete-in-div`") that deletes elements that occur inside of XHTML div elements. It might be used as follows:

```
<ex:delete-in-div xmlns:p="http://www.w3.org/ns/xproc"
                  xmlns:ex="http://example.org/ns/ex"
          xmlns:html="http://www.w3.org/1999/xhtml"
    divchild="html:p[@class='delete']"/>
```

In this case, the `match` option passed to the `p:delete` step needs *both* the namespace binding of "`h`" specified in the `ex:delete-in-div` pipeline definition *and* the namespace binding of "`html`" specified in the `divchild` option on the call of that pipeline. It's not sufficient to provide just one of the sets of bindings.

If pipeline authors cannot arrange for all of the necessary namespace bindings to be in scope, then EQNames can be used to remove the dependency on namespace bindings:

```
<ex:delete-in-div xmlns:p="http://www.w3.org/ns/xproc"
                  xmlns:ex="http://example.org/ns/ex"
    divchild="q{http://www.w3.org/1999/xhtml}p[@class='delete']"/>
```

In this example, the expression will match "`p`" elements in the XHTML namespace irrespective of any bindings that may or may not be in scope.

12. Security Considerations

An XProc pipeline may attempt to access arbitrary network resources: steps such as `p:load` and `p:http-request` can attempt to read from an arbitrary URI; steps such as `p:store` can attempt to write to an arbitrary location; `p:exec` can attempt to execute an arbitrary program. Note, also, that some steps, such as `p:xslt` and `p:xquery`, include extension mechanisms which may attempt to execute arbitrary code.

In some environments, it may be inappropriate to provide the XProc pipeline with access to these resources. In a server environment, for example, it may be impractical to allow pipelines to store data. In environments where the pipeline cannot be trusted, allowing the pipeline to access arbitrary resources or execute arbitrary code may be a security risk.

It is a *dynamic error* (`err:XD0021`) for a pipeline to attempt to access a resource for which it has insufficient privileges or perform a step which is forbidden. Which steps are forbidden, what privileges are needed to access resources, and under what circumstances these security constraints apply is *implementation-dependent*.

Steps in a pipeline may call themselves recursively which could result in pipelines which will never terminate.

A conformant XProc processor may limit the resources available to any or all steps in a pipeline. A conformant implementation may raise dynamic errors, or take any other corrective action, for any security problems that it detects.

13. Versioning Considerations

A pipeline author **may** identify the version of XProc for which a particular pipeline was authored by setting the `version` attribute. The `version` attribute can be specified on `p:declare-step` or `p:library`. If specified, the value of the `version` attribute **must** be a `xs:decimal`. It is a *static error* (`err:XS0063`) if the value of the `version` attribute is not a `xs:decimal`.

The version of XProc defined by this specification is "`3.0`".

A pipeline author **must** identify the version of XProc on the document element of a pipeline document. It is a *static error* (`err:XS0062`) if a required `version` attribute is not present.

The version identified applies to the element on which the `version` attribute appears and all of its descendants, unless or until another version is explicitly identified.

XProc 3.0 takes a draconian approach to versioning. It is a *static error* (`err:XS0060`) if the processor encounters an explicit request for a version of the language other than "3.0".

14. Syntax Overview

This section describes the normative XML syntax of XProc. This syntax is sufficient to represent all the aspects of a pipeline, as set out in the preceding sections. [Definition:

XProc is intended to work equally well with [XML 1.0] and [XML 1.1]. Unless otherwise noted, the term "*XML*" refers equally to both versions.] [Definition: Unless otherwise noted, the term *Namespaces in XML* refers equally to [Namespaces 1.0] and [Namespaces 1.1].] Support for pipeline documents written in XML 1.1 and pipeline inputs and outputs that use XML 1.1 is *implementation-defined*.

Elements in a pipeline document represent the pipeline, the steps it contains, the connections between those steps, the steps and connections contained within them, and so on. Each step is represented by an element; a combination of elements and attributes specify how the inputs and outputs of each step are connected and how options are passed. Outside of inline documents (`p:inline` elements explicitly or implicitly), text nodes that consist entirely of whitespace and XML comments are ignored. XML processing instructions are also generally ignored. It is *implementation-defined* if any processing instructions are significant to an implementation. In an inline document, all markup is treated as if it was a quoted part of the inline document and no special semantics apply except as noted elsewhere in this specification.

Conceptually, we can speak of steps as objects that have inputs and outputs, that are connected together and which may contain additional steps. Syntactically, we need a mechanism for specifying these relationships.

Containment is represented naturally using nesting of XML elements. If a particular element identifies a *compound step* then the step elements that are its immediate children form its *subpipeline*.

The connections between steps are expressed using names and references to those names.

14.1. XProc Namespaces

There are three namespaces associated with XProc:

`http://www.w3.org/ns/xproc`
: The namespace of the XProc XML vocabulary described by this specification; by convention, the namespace prefix "`p:`" is used for this namespace.

`http://www.w3.org/ns/xproc-step`
: The namespace used for documents that are inputs to and outputs from several standard and optional steps described in this specification. Some steps, such as `p:http-request` and `p:store`, have defined input or output vocabularies. We use this namespace for all of those documents. The conventional prefix "`c:`" is used for this namespace.

`http://www.w3.org/ns/xproc-error`
: The namespace used for errors. The conventional prefix "`err:`" is used for this namespace.

This specification also makes use of the prefix "`xs:`" to refer to the [W3C XML Schema: Part 1] namespace `http://www.w3.org/2001/XMLSchema` and the prefix "`xsi:`" to refer to the namespace `http://www.w3.org/2001/XMLSchema-instance`

14.2. Scoping of Names

Names are used to identify step types, steps, ports, options and variables. Step types, options, and variables are named with EQNames. Steps and ports are named with NCNames. The scope of a name is a measure of where it is available in a pipeline. [Definition: If two names are in the same scope, we say that they are *visible* to each other.]

Within a `p:library`, declaring that the `visibility` of a step or static option is "`private`" limits its visibility outside of the library. Note, however, that if other declarations of the same name are visible from the point where the private declaration occurs, that is still an error.

14.2.1. Scoping of step type names

The scope of the names of the step types is the pipeline in which they are declared, including any declarations imported from libraries via `p:import`. Nested pipelines inherit the step types in scope for their parent.

In other words, the step types that are in scope in a `p:declare-step` are:

- The standard, built-in types (`p:declare-step`, `p:choose`, etc.).

- Any implementation-provided types.
- Any step types declared in the `p:declare-step` children of the pipeline element.
- The types of any `p:declare-step`s that are imported.
- Any public types that are in the scope of any `p:library` that is imported.
- Any step types that are in scope for the pipeline's parent `p:declare-step`, if it has one.
- The type of the pipeline itself, if it has one.

The step types that are in scope in a `p:library` are:

- The standard, built-in types (`p:declare-step`, `p:choose`, etc.).
- Any implementation-provided types.
- Any step types declared in the library (the `p:declare-step` children of the `p:library` element).
- The types of `p:declare-step`s that are imported into the library.
- Any public types that are in the scope of any `p:library` that is imported.

All the step types in a pipeline or library **must** have unique names: it is a *static error* (`err:XS0036`) if any step type name is built-in and/or declared or defined more than once in the same scope.

14.2.2. Scoping of step names

The scope of the names of the steps (the values of the step's `name` attributes) is determined by the *environment* of each step. In general, the name of a step, the names of its sibling steps, the names of any steps that it contains directly, the names of its ancestors, and the names of the siblings of its ancestors are all in a common scope. All steps in the same scope **must** have unique names: it is a *static error* (`err:XS0002`) if two steps with the same name appear in the same scope.

14.2.3. Scoping of port names

The scope of an input or output port name is the step on which it is defined. The names of all the ports on any step **must** be unique.

Taken together with the scoping of step names, these uniqueness constraints guarantee that the combination of a step name and a port name uniquely identifies exactly one port on exactly one in-scope step.

14.2.4. Scoping of non-static options and variables

The scope of non-static option and variable names is determined by where they are declared. Their scope consists of the sibling elements that follow its declaration and the descendants of those siblings.

Non-static options and variables declared in parent step declarations are not visible in child step declarations.

14.2.5. Scoping of static option names

The scope of the names of static options is the pipeline in which they are declared, including any declarations imported from libraries via `p:import`. Nested pipelines inherit the static options in scope for their parent.

In other words, the step options that are in scope in a `p:declare-step` are:

- Any static options declared in the step.
- Any public static options that are in the scope of any `p:library` that is imported.
- Any static options that are in scope for the pipeline's parent `p:declare-step`, if it has one.

The static options that are in scope in a `p:library` are:

- Any static options declared in the library (the `p:option` children of the `p:library` element).

- Any public static options that are in the scope of any `p:library` that is imported.

All the static options in a pipeline or library **must** have unique names: it is a *static error* (`err:XS0071`) if any static option name is declared more than once in the same scope.

14.3. Base URIs and xml:base

If a relative URI appears in an option of type `xs:anyURI`, the base URI against which it **must** be made absolute is the base URI of the `p:option` element. If the option value is specified using a syntactic shortcut, the base URI of the step element on which the shortcut attribute appears **must** be used. In general, whenever a relative URI appears in an `xs:anyURI`, its base URI is the base URI of the nearest ancestor element.

The pipeline author can control the base URIs of elements within the pipeline document with the `xml:base` attribute. The `xml:base` attribute **may** appear on any element in a pipeline and has the semantics outlined in [XML Base].

For XML documents, HTML documents, and text documents, the pipeline author can control the base URI of the document node by manipulating the document property "`base-uri`".

14.4. Unique identifiers

A pipeline author can provide a globally unique identifier for any element in a pipeline with the `xml:id` attribute.

The `xml:id` attribute **may** appear on any element in a pipeline and has the semantics outlined in [xml:id].

14.5. Associating Documents with Ports

A document or a sequence of documents can be connected to a port in four ways: *by source*, *by URI*, by providing an *inline document*, or by making it *explicitly empty*. Each

of these mechanisms is allowed where connections may be made, except that `p:input` may not include a connection *by source*.

Specified by URI

[Definition: A document is specified *by URI* if it is referenced with a URI.] The `href` attribute on the `p:document` element is used to refer to documents by URI.

In this example, the input to the `p:identity` step named "`otherstep`" comes from "`http://example.com/input.xml`".

```
<p:output port="result"/>

<p:identity name="otherstep">
  <p:with-input port="source">
    <p:document href="http://example.com/input.xml"/>
  </p:with-input>
</p:identity>
```

See the description of `p:document` for a complete description of how URIs may be specified.

Specified by source

[Definition: A document is specified *by source* if it references a specific port on another step.] The `step` and `port` attributes on the `p:pipe` element are used for this purpose.

In this example, the "`source`" input to the `p:xinclude` step named "`expand`" comes from the "`result`" port of the step named "`otherstep`".

```
<!-- there's no otherstep so this isn't expected to work... -->
<p:xinclude name="expand">
  <p:with-input port="source">
    <p:pipe step="otherstep" port="result"/>
  </p:with-input>
</p:xinclude>
```

See the description of `p:pipe` for a complete description of the ports that can be connected.

Specified inline

[Definition: An *inline document* is specified directly in the body of the element to which it connects.] The content of the `p:inline` element is used for this purpose.

In this example, the "`stylesheet`" input to the XSLT step named "`xform`" comes from the content of the `p:with-input` element itself.

```
<p:xslt name="xform">
  <p:with-input port="stylesheet">
    <p:inline>
      <xsl:stylesheet version="1.0">
        ...
      </xsl:stylesheet>
    </p:inline>
  </p:with-input>
</p:xslt>
```

Inline documents are considered "quoted". The pipeline processor passes them literally to the port, even if they contain elements from the XProc namespace or other namespaces that would have other semantics outside of the `p:inline`.

See the description of `p:inline` for a complete description of how inline documents may be specified.

Specified explicitly empty

[Definition: An *empty sequence* of documents is specified with the `p:empty` element.]

In this example, the "`source`" input to the XSLT 2.0 step named "`generate`" is explicitly empty:

```
<p:xslt name="generate" version="2.0">
  <p:with-input port="source">
    <p:empty/>
  </p:with-input>
  <p:with-input port="stylesheet">
    <p:inline>
      <xsl:stylesheet version="2.0">
        ...
      </xsl:stylesheet>
    </p:inline>
  </p:with-input>
  <p:with-option name="template-name" select="'someName'"/>
</p:xslt>
```

If you omit the connection on a primary input port, a connection to the *default readable port* will be assumed. Making the connection explicitly empty guarantees that the connection will be to an empty sequence of documents.

See the description of `p:empty` for a complete description of empty connections.

Note that a `p:input`, `p:with-input`, or `p:output` element may contain more than one `p:pipe`, `p:document`, or `p:inline` element. If more than one *connection* is provided, then the specified sequence of documents is made available on that port in the same order as the connections.

14.6. Documentation

Pipeline authors may add documentation to their pipeline documents with the `p:documentation` element. Except when it appears as a descendant of `p:inline`, the `p:documentation` element is completely ignored by pipeline processors, it exists simply for documentation purposes. If a `p:documentation` is provided as a descendant of `p:inline`, it has no special semantics, it is treated literally as part of the document to be provided on that port. The `p:documentation` element has no special semantics when it appears in documents that flow through the pipeline.

Pipeline processors that inspect the contents of `p:documentation` elements and behave differently on the basis of what they find are *not conformant*. Processor extensions **must** be specified with `p:pipeinfo`.

14.7. Processor annotations

Pipeline authors may add annotations to their pipeline documents with the `p:pipeinfo` element. The semantics of `p:pipeinfo` elements are *implementation-defined*. Processors **should** specify a way for their annotations to be identified, perhaps with extension attributes.

Where `p:documentation` is intended for human consumption, `p:pipeinfo` elements are intended for processor consumption. A processor might, for example, use annotations to identify some particular aspect of an implementation, to request additional, perhaps non-standard features, to describe parallelism constraints, etc.

When a `p:pipeinfo` appears as a descendant of `p:inline`, it has no special semantics; in that context it **must** be treated literally as part of the document to be provided on that port. The `p:pipeinfo` element has no special semantics when it appears in documents that flow through the pipeline.

14.8. Extension attributes

[Definition: An element from the XProc namespace **may** have any attribute not from the XProc namespace, provided that the expanded-QName of the attribute has a non-null namespace URI. Such an attribute is called an *extension attribute*.]

The presence of an extension attribute must not cause the connections between steps to differ from the connections that would arise in the absence of the attribute. They must not cause the processor to fail to signal an error that would be signaled in the absence of the attribute.

A processor which encounters an extension attribute that it does not implement **must** behave as if the attribute was not present.

14.9. Common Attributes

Several attributes can be used on any XProc step, or even any element in a pipeline. For convenience, they are all summarized here.

Attributes from the XML namespace are allowed anywhere. In particular:

- An `xml:id` attribute is allowed on any element. It has the semantics of [xml:id].
- An `xml:base` attribute is allowed on any element. It has the semantics of [XML Base].
- An `xml:lang` attribute is allowed on any element. It has the semantics of [XML 1.0].
- An `xml:space` attribute is allowed on any element. It has the semantics of [XML 1.0].

The remaining attributes are sometimes in no namespace and sometimes explicitly in the XProc namespace. They are in no namespace when they appear on an XProc element; they are in the XProc namespace when they are on an element in any other namespace. In this way, they do not conflict with the names used in other vocabularies. It is a *static error* (`err:XS0097`) if an attribute in the XProc namespace appears on an element in the XProc namespace.

14.9.1. Expand text attributes

The `[p:]expand-text` and `[p:]inline-expand-text` attributes control whether or not text and attribute nodes in descendant `p:inline` elements and implicit inlines are designated as value templates. Note that they control both text *and* attribute value templates.

The `[p:]expand-text` attribute can appear on all elements in the pipeline. It controls whether or not descendant inlines are designated as value templates. If the attribute *itself* appears among the descendants of a `p:inline` (or implicit inline), then it is a regular attribute and has no special semantics. In this case, the `[p:]inline-expand-text` attribute comes into play.

The `[p:]inline-expand-text` attribute appearing as descendant of a `p:inline` or in an implicit inline is treated as a special attribute, with the same semantics as the `[p:]expand-text` attribute. The attribute will not be part of the result of the `p:inline` or implicit inline.

If the [p:]expand-text or [p:]inline-expand-text attribute appears on more than one element among the ancestors of a text or attribute node in a p:inline element or implicit inline, only the value on the nearest ancestor is considered.

If the nearest [p:]expand-text or [p:]inline-expand-text attribute has the value "false", then the text and attribute nodes in a p:inline element or implicit inline are not value templates. If it has the value "true", or if no such attribute is present among ancestors, then the text and attribute nodes *are* value templates.

Neither [p:]expand-text nor [p:]inline-expand-text are attribute value templates themselves. It is a *static error* (err:XS0113) if either [p:]expand-text or [p:]inline-expand-text is to be interpreted by the processor and it does not have the value "true" or "false".

14.9.2. Conditional Element Exclusion

The [p:]use-when attribute controls whether or not an element (and its descendants) appear in the pipeline. The value of the attribute **must** contain an XPath expression that can be evaluated statically (See Section 11.3, "Static Options".) [Definition: If the effective boolean value of the [p:]use-when expression is false, then the element and all of its descendants are *effectively excluded* from the pipeline document.] If a node is effectively excluded, the processor **must** behave as if the element was not present in the document.

Conditional element exclusion occurs during static analysis of the pipeline.

> **Note**
>
> The effective exclusion of [p:]use-when processing occurs after XML parsing and has no effect on well-formedness or validation errors which will be reported in the usual way.

Deadlock situations can arise if two or more [p:]use-when expressions depend on each other. Consider, for example:

```
<p:declare-step type="ex:A" use-when="p:step-available('ex:B')">
  …
</p:declare-step>

<p:declare-step type="ex:B" use-when="p:step-available('ex:A')">
  …
</p:declare-step>
```

It is not possible for the processor to determine if `ex:A` should be declared without first determining if `ex:B` should be declared, and vice versa.

It is a *static error* (`err:XS0115`) if two or more elements are contained within a deadlocked network of `[p:]use-when` expressions. In order to avoid deadlock, there must exist an order in which every expression can be resolved without reference to an expression that occurs after it in the ordering.

Processors may be required to evaluate expressions in an arbitrary order, but they are not required to solve a set of linear equations simultaneously. So, while "declare both `ex:A` and `ex:B`" is a valid solution to the example above, conformant processors are not required (or allowed) to find it because neither the order "A then B" nor the order "B then A" is sufficient to find the solution.

14.9.3. Additional dependent connections

The `[p:]depends` attribute can appear on any step invocation *except* `p:when`, `p:otherwise`, `p:catch`, and `p:finally`. It adds an explicit dependency between steps. The value of the attribute is a space separated list of step names. It is a *static error* (`err:XS0073`) if any specified name is not the name of an in-scope step.

In most pipelines, the dependencies that arise naturally from the connections between steps are sufficient. If step "B" consumes the output of step "A", then clearly "A" must run before "B". However, it is sometimes the case that one step depends on another in ways that are not apparent in the connections. Consider, for example, a pipeline that interacts with two different web services. It may very well be the case that one web service has to run before the other, even though the latter does not consume any output from the former.

When `[p:]depends` is used, if step “Y” depends on step “X”, then “X” must run before “Y”.

The connections specified by the `[p:]depends` attribute apply *in addition to* the dependencies that arise naturally from connections between steps. Taken together with the input and output connections, the graph must not contain any loops.

The `[p:]depends` attribute is forbidden from several elements because they are only conditionally evaluated. The semantics of dependency are ambiguous at best in this case. Moving the dependency to the parent element resolves this ambiguity.

14.9.4. Controlling long running steps

The `[p:]timeout` attribute allows a pipeline author to suggest a length of time beyond which the pipeline processor should consider that a step has taken an excessive amount of time.

The value of the `[p:]timeout` option must be a `xs:nonNegativeInteger`. It is interpreted as a number of seconds. The value zero may be used to indicate that no limit is expressed (this is the same as omitting the attribute, but may sometimes be more convenient for pipeline authors).

It is a *dynamic error* (`err:XD0053`) if a step runs longer than its timeout value.

The precise amount of time a step takes to perform its task depends on many factors (the hardware running the processor, the processor’s execution strategy, the system load etc.) This feature can not be used as an exact timing tool in XProc. Developers are advised to calculate the value for `[p:]timeout` generously, so the dynamic error is raised only in extreme cases.

It is *implementation-defined* whether a processor supports timeouts, and if it does, how precisely and precisely how the execution time of a step is measured.

14.9.5. Status and debugging output

The `[p:]message` attribute can appear on any step invocation. It’s value is treated as an attribute value template (irrespective of any enclosing `[p:]expand-text` setting) and the computed value is made available.

Precisely what "made available" means is *implementation-defined*. It will often be as simple as printing the message on some output channel. But for embedded systems or other environments where "print it for the user" is meaningless or inconvenient, some other mechanism may be used.

If a processor can make the message available, it **should** do so before execution of the step begins.

14.10. Syntax Summaries

The description of each element in the pipeline namespace is accompanied by a syntactic summary that provides a quick overview of the element's syntax:

```
<p:some-element
  reqd-attribute = some-type
  some-attribute? = some-type
  avt-attribute? = { some-type }>
    (some |
     elements |
     allowed)*,
    other-elements?
</p:some-element>
```

The content model fragments in these tableaux are presented in a simple, compact notation. In brief:

Attributes

- Required attributes are bold. Optional attributes are followed by a question mark.
- If an attribute value may contain an attribute value template, its type is shown in curly brackets: "`{ some-type }`". If `some-type` is `xs:anyURI`, `xs:QName`, or a map type with key type of `xs:QName`, the conversions described in Section 11.5, "Implicit casting" apply.
- An attribute value with a map type marks an `XPathExpression` expected to deliver a map of the indicated type. If the map type has a key type of `xs:QName`, the conversions described in Section 11.5, "Implicit casting" apply.

Elements

- A name represent exactly one occurrence of an element with that name.
- Parentheses are used for grouping.
- Elements or groups separated by a comma (“,”) represent an ordered sequence: a followed by b followed by c: (a,b,c).
- Elements or groups separated by a vertical bar (“|”) represent a choice: a or b or c: (a | b | c).
- Elements or groups separated by an ampersand (“&”) represent an unordered sequence: a and b and c, in any order: (a & b & c).
- An element or group followed by a question mark (“?”) is optional; it may or may not occur but if it occurs it can occur only once.
- An element or group followed by an asterisk (“*”) is optional and may be repeated; it may or may not occur and if it occurs it can occur any number of times.
- An element or group followed by a plus (“+”) is required and may be repeated; it must occur at least once, and it can occur any number of times.

For clarity of exposition, the common attributes (see Section 14.9, “Common Attributes”) are elided from the summaries as are the `p:documentation` and `p:pipeinfo` elements, which are allowed anywhere, and attributes that are syntactic shortcuts for option values.

The types given for attributes should be understood as follows:

- `ID`, `NCName`, `NMTOKEN`, `NMTOKENS`, `anyURI`, `boolean`, `integer`, `string`: As per [W3C XML Schema: Part 2] including whitespace normalization as appropriate.
- `EQName`: With whitespace normalization as per [W3C XML Schema: Part 2] for QNames. Note, however, that QNames that have no prefix are always in no-namespace, irrespective of the default namespace.
- `EQNameList`: As a whitespace separated list of EQNames, per the definition above.

- `PrefixList`: As a list with `[item type] NMTOKEN`, per [W3C XML Schema: Part 2], including whitespace normalization.
- `ExcludeInlinePrefixes`: As a `PrefixList` per the definition above, with the following extensions: the tokens `#all` and `#default` may appear.
- `XPathExpression`, `XSLTSelectionPattern`: As a string per [W3C XML Schema: Part 2], including whitespace normalization, and the further requirement to be a conformant Expression per [XPath 3.1] or *selection pattern* per [XSLT 3.0].
- `MediaTypes`: As a whitespace separated list of media types as defined in [RFC 2046].

14.11. Common errors

A number of errors apply generally:

- It is a *static error* (`err:XS0059`) if the pipeline element is not `p:declare-step` or `p:library`.
- It is a *static error* (`err:XS0008`) if any element in the XProc namespace has attributes not defined by this specification unless they are *extension attributes*.
- It is a *static error* (`err:XS0038`) if any required attribute is not provided.
- It is a *static error* (`err:XS0077`) if the value on an attribute of an XProc element does not satisfy the type required for that attribute.
- It is a *dynamic error* (`err:XD0028`) if any attribute value does not satisfy the type required for that attribute.
- It is a *static error* (`err:XS0044`) if any step contains an atomic step for which there is no visible declaration.
- It is a *static error* (`err:XS0037`) if any user extension step or any element in the XProc namespace other than `p:inline` directly contains text nodes that do not consist entirely of whitespace.
- It is a *static error* (`err:XS0015`) if a compound step has no *contained steps*.

- It is a *dynamic error* (`err:XD0012`) if any attempt is made to dereference a URI where the scheme of the URI reference is not supported. Implementations are encouraged to support as many schemes as is practical and, in particular, they **should** support both the `file:` and `http(s):` schemes. The set of URI schemes actually supported is *implementation-defined*.

- It is a *dynamic error* (`err:XD0030`) if a step is unable or incapable of performing its function. This is a general error code for "step failed" (e.g., if the input isn't of the expected type or if attempting to process the input causes the implementation to abort). Users and implementers who create extension steps are encouraged to use this code for general failures.

- In most steps which use a select expression or *selection pattern*, any kind of node can be identified by the expression or pattern. However, some expressions and patterns on some steps are only applicable to some kinds of nodes (e.g., it doesn't make sense to speak of adding attributes to a comment!).

  It is a *dynamic error* (`err:XC0023`) if a select expression or *selection pattern* returns a node type that is not allowed by the step.

- It is a *static error* (`err:XS0100`) if the pipeline document does not conform to the grammar for pipeline documents. This is a general error code indicating that the pipeline is syntactically incorrect in some way not identified more precisely in this specification.

If an XProc processor can determine statically that a dynamic error will *always* occur, it **may** report that error statically provided that the error *does not* occur among the descendants of a `p:try`. Dynamic errors inside a `p:try` **must not** be reported statically. They must be raised dynamically so that `p:catch` processing can be performed on them.

15. Steps

This section describes the core language steps of XProc; the full vocabulary of standard, atomic steps is described in [Steps 3.0].

15.1. Pipelines

The document element of a pipeline document is `p:declare-step` which declares a pipeline that can be evaluated by an XProc processor.

It encapsulates the behavior of a *subpipeline*. Its children declare inputs, outputs, and options that the pipeline exposes and identify the steps in its subpipeline.

Viewed from the outside, a `p:declare-step` is a black box which performs some calculation on its inputs and produces its outputs. From the pipeline author's perspective, the computation performed by the pipeline is described in terms of *contained steps* which read the pipeline's inputs and produce the pipeline's outputs.

A `p:declare-step` element can also be nested inside other `p:declare-step` or `p:library` elements in which case it simply declares a pipeline that will be run elsewhere.

For more details, see Section 16.5, "p:declare-step".

15.1.1. Example

A pipeline might accept a document as input; perform XInclude, validation, and transformation; and produce the transformed document as its output.

Example 4. A Sample Pipeline Document

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                version="3.0">
<p:input port="source"/>
<p:output port="result"/>

<p:xinclude/>

<p:validate-with-xml-schema>
  <p:with-input port="schema">
    <p:document href="http://example.com/path/to/schema.xsd"/>
  </p:with-input>
</p:validate-with-xml-schema>

<p:xslt>
  <p:with-input port="stylesheet">
    <p:document href="http://example.com/path/to/stylesheet.xsl"/>
  </p:with-input>
</p:xslt>

</p:declare-step>
```

15.2. p:for-each

A for-each is specified by the `p:for-each` element. It is a *compound step* that processes a sequence of documents, applying its *subpipeline* to each document in turn.

```
<p:for-each
  name? = NCName>
    ((p:with-input? &
      p:output*),
     subpipeline)
</p:for-each>
```

When a pipeline needs to process a sequence of documents using a subpipeline that only processes a single document, the `p:for-each` construct can be used as a wrapper around that subpipeline. The `p:for-each` will apply that subpipeline to each document in the sequence in turn.

The result of the `p:for-each` is a sequence of documents produced by processing each individual document in the input sequence. If the `p:for-each` has one or more output ports, what appears on each of those ports is the sequence of documents that is the concatenation of the sequence produced by each iteration of the loop on the port to which it is connected. If the iteration source for a `p:for-each` is an empty sequence, then the subpipeline is never run and an empty sequence is produced on all of the outputs.

The `p:for-each` has a single *anonymous input*: its *connection* is provided by the `p:with-input`. If no iteration sequence is explicitly provided, then the iteration source is read from the *default readable port*.

The processor provides each document, one at a time, to the *subpipeline* represented by the children of the `p:for-each` on a port named `current`.

For each declared output, the processor collects all the documents that are produced for that output from all the iterations, in order, into a sequence. The result of the `p:for-each` on that output is that sequence of documents.

The environment inherited by the *contained steps* of a `p:for-each` is the *inherited environment* with these modifications:

- The port named "`current`" on the `p:for-each` is added to the *readable ports*.
- The port named "`current`" on the `p:for-each` is made the *default readable port*.

If the `p:for-each` has a *primary output port* (explicit or supplied by default) and that port has no *connection*, then it is connected to the *primary output port* of the *last step* in the *subpipeline*. It is a *static error* (`err:XS0006`) if the primary output port has no explicit connection and the *last step* in the subpipeline does not have a primary output port.

Note that outputs declared for a `p:for-each` serve a dual role. Inside the `p:for-each`, they are used to read results from the subpipeline. Outside the `p:for-each`, they provide the aggregated results.

The `sequence` attribute on a `p:output` inside a `p:for-each` only applies inside the step. From the outside, all of the outputs produce sequences.

15.2.1. XPath Context

Within a `p:for-each`, the `p:iteration-position` and `p:iteration-size` are taken from the sequence of documents that will be processed by the `p:for-each`. The total number of documents is the `p:iteration-size`; the ordinal value of the current document (the document appearing on the `current` port) is the `p:iteration-position`.

> **Note to implementers**
>
> In the case where no XPath expression that must be evaluated by the processor makes any reference to `p:iteration-size`, its value does not actually have to be calculated (and the entire input sequence does not, therefore, need to be buffered so that its size can be calculated before processing begins).

15.2.2. Example

A `p:for-each` might accept a sequence of chapters as its input, process each chapter in turn with XSLT, a step that accepts only a single input document, and produce a sequence of formatted chapters as its output.

Example 5. A Sample For-Each

```
<p:for-each name="chapters">
  <p:with-input select="//chapter"/>
  <p:output port="html-results">
    <p:pipe step="make-html" port="result"/>
  </p:output>
  <p:output port="fo-results">
    <p:pipe step="make-fo" port="result"/>
  </p:output>

  <p:xslt name="make-html">
    <p:with-input port="stylesheet"
                  href="http://example.com/xsl/html.xsl"/>
  </p:xslt>

  <p:xslt name="make-fo">
    <p:with-input port="source" pipe="current@chapters"/>
    <p:with-input port="stylesheet"
                  href="http://example.com/xsl/fo.xsl"/>
  </p:xslt>
</p:for-each>
```

The `//chapter` elements of the document are selected. Each chapter is transformed into HTML and XSL Formatting Objects using an XSLT step. The resulting HTML and FO documents are aggregated together and appear on the `html-results` and `fo-results` ports, respectively, of the `chapters` step itself.

15.3. p:viewport

A viewport is specified by the `p:viewport` element. It is a *compound step* that processes single XML or HTML documents, applying its *subpipeline* to one or more subtrees of each document in turn.

```
<p:viewport
  name? = NCName
  match = XSLTSelectionPattern>
    ((p:with-input? &
      p:output?),
     subpipeline)
</p:viewport>
```

The result of the `p:viewport` is a copy of the original document where the selected subtrees have been replaced by the results of applying the subpipeline to them.

The `p:viewport` has a single *anonymous input*: its *connection* is provided by the `p:with-input`. If no document is explicitly provided, then the viewport source is read from the *default readable port*. If the `p:viewport` input is a sequence, each document in the sequence is processed in turn producing a sequence on the output. It is a *dynamic error* (`err:XD0072`) if a document appearing on the input port of `p:viewport` is neither an XML document nor an HTML document.

The `match` attribute specifies an XSLT *selection pattern*. Each matching node in the source document is wrapped in a document node, as necessary, and provided, one at a time, to the viewport's *subpipeline* on a port named `current`. The base URI of the resulting document that is passed to the subpipeline is the base URI of the matched node. It is a *dynamic error* (`err:XD0010`) if the `match` expression on `p:viewport` matches an attribute or a namespace node.

> **Note**
>
> The `match` attribute on `p:viewport` is a selection pattern and may contain references to in-scope variables and options, but it is not an *attribute value template*.

After a match is found, the entire subtree rooted at that match is processed as a unit. No further attempts are made to match nodes among the descendants of any matched node.

The environment inherited by the *contained steps* of a `p:viewport` is the *inherited environment* with these modifications:

- The port named "current" on the p:viewport is added to the *readable ports*.
- The port named "current" on the p:viewport is made the *default readable port*.

The p:viewport must contain a single, *primary output port* declared explicitly or supplied by default. If that port has no *connection*, then it is connected to the *primary output port* of the *last step* in the *subpipeline*. It is a *static error* (err:XS0006) if the primary output port is unconnected and the *last step* in the subpipeline does not have a primary output port.

What appears on the output from the p:viewport will be a copy of the input document where each matching node is replaced by the result of applying the subpipeline to the subtree rooted at that node. In other words, if the *selection pattern* matches a particular node then that node is wrapped in a document node and provided on the current port, the subpipeline in the p:viewport is evaluated, and the result that appears on the output port replaces the matched node.

If a document resulting from applying the subpipeline to the matched node is an XML document, an HTML document, or a text document, all child nodes of the document node will be used to replace the matched node. It is a *dynamic error* (err:XD0073) if the document returned by applying the subpipeline to the matched node is not an XML document, an HTML document, or a text document.

If no documents appear on the output port, the matched node will effectively be deleted. If exactly one document appears, the contents of that document will replace the matched node. If a sequence of documents appears, then the contents of each document in that sequence (in the order it appears in the sequence) will replace the matched node.

The output of the p:viewport itself is a sequence of documents that appear on a port named "result". Note that the semantics of p:viewport are special. The output port in the p:viewport is used only to access the results of the subpipeline. The output of the step itself appears on a port with the fixed name "result" that is never explicitly declared.

For the documents appearing on port result all document properties will be preserved, except when option match matches a document node and the result from applying the subpipeline to the document node is a (sequence of) text document(s).

In this case the content-type property is changed to "`text/plain`" and the serialization property is removed, while all other document properties are preserved.

15.3.1. XPath Context

Within a `p:viewport`, the `p:iteration-position` and `p:iteration-size` are taken from the sequence of documents that will be processed by the `p:viewport`. The total number of documents is the `p:iteration-size`; the ordinal value of the current document (the document appearing on the `current` port) is the `p:iteration-position`.

> **Note to implementers**
>
> In the case where no XPath expression that must be evaluated by the processor makes any reference to `p:iteration-size`, its value does not actually have to be calculated (and the entire input sequence does not, therefore, need to be buffered so that its size can be calculated before processing begins).

15.3.2. Example

A `p:viewport` might accept an XHTML document as its input, add an `hr` element at the beginning of all `div` elements that have the class value "chapter", and return an XHTML document that is the same as the original except for that change.

Example 6. A Sample Viewport

```
<p:viewport match="h:div[@class='chapter']"
            xmlns:h="http://www.w3.org/1999/xhtml">
  <p:insert position="first-child">
    <p:with-input port="insertion">
      <hr xmlns="http://www.w3.org/1999/xhtml"/>
    </p:with-input>
  </p:insert>
</p:viewport>
```

The nodes which match `h:div[@class='chapter']` in the input document are selected. An `hr` is inserted as the first child of each `h:div` and the resulting version replaces the original `h:div`. The result of the whole step is a copy of the input document with a horizontal rule as the first child of each selected `h:div`.

15.4. p:choose

A choose step is specified by the `p:choose` element. It is a *compound step* that contains several, alternate subpipelines. One subpipeline is selected based on the evaluation of XPath expressions.

```
<p:choose
  name? = NCName>
    (p:with-input?,
     ((p:when+,
       p:otherwise?) |
      (p:when*,
       p:otherwise)))
</p:choose>
```

A `p:choose` contains an arbitrary number of alternative *subpipelines*, at most one of which will be evaluated. It is a *static error* (`err:XS0074`) if a `p:choose` has neither a `p:when` nor a `p:otherwise`.

The list of alternative subpipelines consists of zero or more subpipelines guarded by an XPath expression, followed optionally by a single default subpipeline.

The `p:choose` considers each subpipeline in turn and selects the first (and only the first) subpipeline for which the guard expression evaluates to true in its context. After a subpipeline is selected, no further guard expressions are evaluated. If there are no subpipelines for which the expression evaluates to true then, if a default subpipeline was specified, it is selected, otherwise, no subpipeline is selected.

After a *subpipeline* is selected, it is evaluated as if only it had been present.

The outputs of the `p:choose` are taken from the outputs of the selected *subpipeline*. The outputs *available* from the `p:choose` are union of all of the outputs declared in any of its alternative subpipelines. In order to maintain consistency with respect to the *default readable port*, if any subpipeline has a *primary output port*, even implicitly, then *every* subpipline must have a primary output port with the same name. In some cases, this may require making the implicit primary output explicit in order to assure that it has the same name. It is a *static error* (`err:XS0102`) if alternative subpipelines have different primary output ports.

Consider a `p:choose` that has two alternative subpipelines where one declares output ports “A” and “B” and the other declares output ports “B” and “C”. The outputs available from the `p:choose` are “A”, “B”, and “C”. No documents appear on any outputs not declared in the subpipline actually selected.

As a convenience to authors, it is not an error if some subpipelines declare outputs that can produce sequences and some do not. Each output of the `p:choose` is declared to produce a sequence. The content types that can appear on the port are the union of the content types that might be produced by any of the `p:when` or the `p:otherwise`. If a primary output port is (explicitly or implicitly) defined and `p:otherwise` is missing, documents with *any* content type can appear on that port.

The `p:choose` can specify the context item against which the XPath expressions that occur on each branch are evaluated. The context item is specified as a *connection* in the `p:with-input`. If no explicit connection is provided, the *default readable port* is used. If the context item is connected to `p:empty`, or is connected to more than one document, or is unconnected and the *default readable port* is undefined, the context item is undefined. It is a *dynamic error* (`err:XD0001`) if an XPath expression makes reference to the context item, size, or position when the context item is undefined.

Each conditional *subpipeline* is represented by a `p:when` element. The default branch is represented by a `p:otherwise` element. These elements are not sibling steps in the usual sense, the names of sibling `p:when` elements and the `p:otherwise` element are not in the same scope.

If the following conditions apply:

- The `p:choose` does not contain a `p:otherwise` child element
- The `p:when` branches all define a primary output port (either implicitly or explicitly)
- None of the effective boolean values of the `p:when` test expressions evaluates to `true`

Then the `p:choose` copies any documents that appear on its default readable port to its primary output port. No documents will be written to the primary output port if there isn't a default readable port, but that is not an error in this case. No documents will ever be written to any non-primary output ports in this case.

Informally: the default sub-pipeline for a missing `p:otherwise` is a `p:identity` step (with the additional feature that it isn't an error if there's no default readable port). If the `p:when` branches do not have a primary output port, no output will be produced on any port.

15.4.1. p:when

A when specifies one subpipeline guarded by a test expression.

```
<p:when
  name? = NCName
  test = XPathExpression
  collection? = boolean>
    (p:with-input?,
     p:output*,
     subpipeline)
</p:when>
```

Each `p:when` branch of the `p:choose` has a `test` attribute which **must** contain an XPath expression. That XPath expression's effective boolean value is the guard for the *subpipeline* contained within that `p:when`.

The `p:when` can specify a context item against which its `test` expression is to be evaluated. That context item is specified as a *connection* for the `p:with-input`. If no context is specified on the `p:when`, the context of the `p:choose` is used. The context item is undefined if the connection or the context of the `p:choose` provides no or more than one document. It is a *dynamic error* (`err:XD0001`) if an XPath expression makes reference to the context item, size, or position when the context item is undefined.

If the `collection` attribute has the value true, then the default collection will contain all of the documents that appeared on that input and the context item will be undefined.

15.4.2. p:otherwise

An otherwise specifies the default branch; the subpipeline selected if no test expression on any preceding `p:when` evaluates to true.

```
<p:otherwise
  name? = NCName>
    (p:output*,
     subpipeline)
</p:otherwise>
```

15.4.3. Example

A `p:choose` might test the version attribute of the document element and validate with an appropriate schema.

Example 7. A Sample Choose

```
<p:choose name="version">
  <p:when test="/*[@version = 2]">
    <p:validate-with-xml-schema>
      <p:with-input port="schema" href="v2schema.xsd"/>
    </p:validate-with-xml-schema>
  </p:when>

  <p:when test="/*[@version = 1]">
    <p:validate-with-xml-schema>
      <p:with-input port="schema" href="v1schema.xsd"/>
    </p:validate-with-xml-schema>
  </p:when>

  <p:when test="/*[@version]">
    <p:identity/>
  </p:when>

  <p:otherwise>
    <p:error code="NOVERSION">
      <p:with-input port="source">
    <p:inline>
      <message>Required version attribute missing.</message>
    </p:inline>
      </p:with-input>
    </p:error>
  </p:otherwise>
</p:choose>
```

15.5. p:if

A `p:if` specifies a single subpipeline guarded by a test expression.

```
<p:if
  name? = NCName
  test = XPathExpression
  collection? = boolean>
    (p:with-input?,
     p:output*,
     subpipeline)
</p:if>
```

The `p:if` step has a `test` attribute which **must** contain an XPath expression. That XPath expression's effective boolean value is the guard for the *subpipeline* contained within it.

The `p:if` step can specify a context item against which its `test` expression is to be evaluated. That context node is specified as a *connection* for the `p:with-input`. If no context is specified on the `p:if`, the context comes from the *default readable port*. If no context is specified and there is no default readable port, the context item is undefined. The context item is also undefined, if no or more than one document is provided. It is a *dynamic error* (`err:XD0001`) if an XPath expression makes reference to the context item, size, or position when the context item is undefined.

If the `collection` attribute has the value true, then the default collection will contain all of the documents that appeared on that input and the context item will be undefined.

The `p:if` must specify a primary output port (either implicitly or explicitly). It is a *static error* (`err:XS0108`) if the `p:if` step does not specify a primary output port.

The requirement for a primary output port stems from the semantics of `p:if`:

- If the effective boolean value of the test expression is true, then the subpipline will be run. The output of the `p:if` in this case is determined by the output ports of the step and what the subpipeline sends to them.
- If the effective boolean value of the test expression is false, then `p:if` copies any documents that appear on its default readable port to its primary output port. No documents will be written to the primary output port if there isn't a default readable port, but that is not an error in this case. No documents will ever be written to any non-primary output ports if the test expression is false.

Informally, if the test expression is false, then `p:if` acts like the identity step (with the additional feature that it isn't an error if there's no default readable port). A primary output port is required in order to make these semantics meaningful and consistent.

The `sequence` attribute and the `content-types` attribute of the primary output port only apply to the subpipeline of `p:if`. From the outside a primary output port of a `p:if` produces sequences and allows documents of any content type. For all other

output ports the `sequence` attribute only applies to the subpipeline, while on the outside these ports may produce sequences.

15.6. p:group

A group is specified by the `p:group` element. It is a *compound step* that encapsulates the behavior of its *subpipeline*.

```
<p:group
  name? = NCName>
    (p:output*,
     subpipeline)
</p:group>
```

A `p:group` is a convenience wrapper for a collection of steps.

15.6.1. Example

Example 8. An Example Group

```
<p:group>
  <p:variable name="db-key"
          select="'some-long-string-of-nearly-random-characters'"/>

  <p:choose>
    <p:when test="/config/output = 'fo'">
      <p:xslt>
    <p:with-option name="parameters" select="map {'key': $db-key }"/>
    <p:with-input port="stylesheet" href="fo.xsl"/>
      </p:xslt>
    </p:when>
    <p:when test="/config/output = 'svg'">
      <p:xslt>
    <p:with-option name="parameters" select="map {'key': $db-key }"/>
    <p:with-input port="stylesheet" href="svg.xsl"/>
      </p:xslt>
    </p:when>
    <p:otherwise>
      <p:xslt>
    <p:with-option name="parameters" select="map {'key': $db-key }"/>
    <p:with-input port="stylesheet" href="html.xsl"/>
      </p:xslt>
    </p:otherwise>
  </p:choose>
</p:group>
```

15.7. p:try

A try/catch step is specified by the `p:try` element. It is a *compound step* that isolates its initial subpipeline, preventing dynamic errors that arise within it from being exposed to the rest of the pipeline. The `p:try` includes alternate recovery subpipelines, and may include a “finally” subpipeline to perform post-processing irrespective of the outcome of the `p:try`.

```
<p:try
  name? = NCName>
    (p:output*,
     subpipeline,
     ((p:catch+,
       p:finally?) |
      (p:catch*,
       p:finally)))
</p:try>
```

The step begins with the initial subpipeline; the recovery (or "catch") pipelines are identified with `p:catch` elements; a "finally" pipeline is identified with a `p:finally` element.

It is a *static error* (`err:XS0075`) if a `p:try` does not have at least one subpipeline step, at least one of `p:catch` or `p:finally`, and at most one `p:finally`.

The `p:try` step evaluates the initial subpipeline and, if no errors occur, the outputs of that pipeline are the outputs of the `p:try` step. However, if any errors occur, the `p:try` abandons the first subpipeline, discarding any output that it might have generated, and considers the recovery subpipelines. If there is no matching recovery subpipeline, the `p:try` fails.

> **Note**
>
> If the initial subpipeline fails, none of its outputs will be visible outside of the `p:try`, but it's still possible for steps in the partially evaluated pipeline to have side effects that are visible outside the processor. For example, a web server might record that some interaction was performed, or a file on the local file system might have been modified.

If a recovery subpipeline is evaluated, the outputs of the recovery subpipeline are the outputs of the `p:try` step. If the recovery subpipeline is evaluated and a step within that subpipeline fails, the `p:try` fails.

Irrespective of whether the initial subpipeline succeeds or fails, if any recovery pipeline is selected, and whether it succeeds or fails, the `p:finally` block is *always* run after all other processing of the `p:try` has finished.

The outputs of the `p:try` are taken from the outputs of the initial *subpipeline* or the recovery subpipline if an error occurred in the initial subpipeline. The outputs *available* from the `p:try` are union of all of the outputs declared (explicitly or implicitly in the absence of any `p:output` elements if the *last step* has a primary output port) in any of its alternative subpipelines. In order to maintain consistency with respect to the *default readable port*, if any subpipeline has a *primary output port*, even implicitly, then *every* subpipline must have a primary output port with the same name. In some cases, this may require making the implicit primary output explicit in order to assure that it has the same name. It is a *static error* (`err:XS0102`) if alternative subpipelines have different primary output ports.

Consider a `p:try` that has an initial subpipeline that declares output ports “A” and “B” and a recovery subpipeline that declares output ports “B” and “C”. The outputs available from the `p:try` are “A”, “B”, and “C”. No documents appear on any outputs not declared in the subpipeline whose results are actually returned.

As a convenience to authors, it is not an error if an output port can produce a sequence in the initial subpipeline but not in the recovery subpipeline, or vice versa. Each output of the `p:try` is declared to produce a sequence. The content types that can appear on the port are the union of the content types that might be produced by the initial subpipeline and any of the recovery subpipelines.

A pipeline author can cause an error to occur with the `p:error` step.

If we assume that an absent `p:finally` always succeeds, evaluation of a `p:try` falls into one of these cases:

- If the initial pipeline succeeds:
    - If the `p:finally` succeeds, the `p:try` succeeds and the outputs of the initial subpipeline are the outputs of the `p:try`.
    - If the `p:finally` fails, the `p:try` fails and the error raised by the `p:finally` is reported as the cause of the failure.

- If the initial pipeline fails and a recovery subpipeline is selected:
  - If the recovery pipeline succeeds:
    - If the `p:finally` succeeds, the `p:try` succeeds and the outputs of the recovery subpipeline are the outputs of the `p:try`.
    - If the `p:finally` fails, the `p:try` fails and the error raised by the `p:finally` is reported as the cause of the failure.
  - If the recovery pipeline fails:
    - If the `p:finally` succeeds, the `p:try` fails and the error raised by the recovery subpipeline is reported as the cause of the failure.
    - If the `p:finally` fails, the `p:try` fails and the error raised by the *recovery subpipeline* **must** be reported as the cause of the failure. The error raised by the finally pipeline **may** also be reported in addition to the error raised by the recovery pipeline.
- If the initial pipeline fails and a recovery subpipeline is not selected:
  - If the `p:finally` succeeds, the `p:try` fails and the error raised by the initial subpipeline is reported as the cause of the failure.
  - If the `p:finally` fails, the `p:try` fails and the error raised by the *initial subpipeline* **must** be reported as the cause of the failure. The error raised by the finally pipeline **may** also be reported in addition to the error raised by the initial subpipeline.

The `p:catch` and `p:finally` elements are not sibling steps, the names of sibling `p:catch` elements and the `p:finally` element are not in the same scope. The elements of the initial subpipeline are also not in the same scope as the `p:catch` and `p:finally` elements or their descendants.

15.7.1. p:catch

A `p:catch` is a recovery subpipeline.

```
<p:catch
  name? = NCName
  code? = EQNameList>
    (p:output*,
     subpipeline)
</p:catch>
```

The environment inherited by the *contained steps* of the `p:catch` is the *inherited environment* with these modifications:

- A primary input port named "`error`" is added to the *readable ports* on the `p:catch`.
- Output ports and variables from the `p:try`'s subpipeline are not available.

All except the last `p:catch` pipeline **must** have a `code` attribute. It is a *static error* (`err:XS0064`) if the `code` attribute is missing from any but the last `p:catch` or if any error code occurs in more than one `code` attribute among sibling `p:catch` elements. It is a *static error* (`err:XS0083`) if the value of the `code` attribute is not a whitespace separated list of EQNames.

When a `p:try` considers the recovery subpipelines, if any of the specified error codes in a `p:catch` match the error that was raised in the initial subpipeline, then that `p:catch` is selected as the recovery pipeline. If the last `p:catch` does not have a `code` attribute, it is selected if no other `p:catch` has a matching error code.

What appears on the `error` input port is an error document. The error document may contain messages generated by steps that were part of the initial subpipeline. Not all messages that appear are indicative of errors; for example, it is common for all `xsl:message` output from the XSLT component to appear on the `error` input port. It is possible that the component which fails may not produce any messages at all. It is also possible that the failure of one component may cause others to fail so that there may be multiple failure messages in the document.

15.7.2. p:finally

The last thing that the `p:try` step does is evaluate the `p:finally` pipeline.

```
<p:finally
  name? = NCName>
    (p:output*,
     subpipeline)
</p:finally>
```

The environment inherited by the *contained steps* of the `p:finally` is the *inherited environment* with these modifications:

- A primary input port named "`error`" is added to the *readable ports* on the `p:finally`.
- Output ports and variables from the `p:try`'s subpipeline are not available.

If no error occurred, there will be no documents on the `error` port.

The `p:finally` exists only to handle recovery and resource cleanup tasks. Because the `p:finally` will always be evaluated, it must not have output ports that might conflict with the output ports of either the initial subpipline or any `p:catch`. It is a *static error* (`err:XS0072`) if the name of any output port on the `p:finally` is the same as the name of any other output port in the `p:try` or any of its sibling `p:catch` elements. It is a *static error* (`err:XS0112`) if `p:finally` declares a primary output port either explicitly or implicitly.

15.7.3. The Error Vocabulary

In general, it is very difficult to predict error behavior. Step failure may be catastrophic (programmer error), or it may be the result of user error, resource failures, etc. Steps may detect more than one error, and the failure of one step may cause other steps to fail as well.

The `p:try`/`p:catch` mechanism gives pipeline authors the opportunity to process the errors that caused the `p:try` to fail. In order to facilitate some modicum of interoperability among processors, errors that are reported on the `error` input port of a `p:catch` **should** conform to the format described here.

15.7.3.1. c:errors

The error vocabulary consists of a root element, `c:errors` which contains zero or more `c:error` elements.

```
<c:errors>
    c:error*
</c:errors>
```

15.7.3.2. c:error

Each specific error is represented by an `c:error` element:

```
<c:error
  name? = NCName
  type? = EQName
  code? = EQName
  cause? = EQName
  href? = anyURI
  line? = integer
  column? = integer
  offset? = integer>
    anyNode*
</c:error>
```

The `name` and `type` attributes identify the name and type, respectively, of the step which failed.

The `code` is an EQName which identifies the error. For steps which have defined error codes, this is an opportunity for the step to identify the error in a machine-processable fashion. Many steps omit this because they do not include the concept of errors identified by EQNames.

The `cause` is an EQName which identifies an underlying error, if applicable. As an aide to interoperability, this specification mandates particular error codes for conditions that can arise in a variety of ways. For example, `err:XD0050` is raised for all errors in XPath expressions in value templates. The implementation may use `cause` to record an underlying error (for example, the XPath error code). The error codes that appear in `cause` are *implementation-dependent*.

If the error was caused by a specific document, or by the location of some erroneous construction in a specific document, the `href`, `line`, `column`, and `offset` attributes identify this location. Generally, the error location is identified either with line and column numbers or with an offset from the beginning of the document, but not usually both.

The content of the `c:error` element is any well-formed XML. Specific steps, or specific implementations, may provide more detail about the format of the content of an error message.

15.7.3.3. Error Example

Consider the following XSLT stylesheet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:template match="/">
  <xsl:message terminate="yes">
    <xsl:text>This stylesheet is </xsl:text>
    <emph>pointless</emph>
    <xsl:text>.</xsl:text>
  </xsl:message>
</xsl:template>

</xsl:stylesheet>
```

If it was used in a step named “xform” in a `p:try`, the following error document might be produced:

```
<c:errors xmlns:c="http://www.w3.org/ns/xproc-step">
  <c:error name="xform" type="p:xslt"
             href="style.xsl" line="6">This stylesheet is
<emph>pointless</emph>.</c:error>
</c:errors>
```

It is not an error for steps to generate non-standard error output as long as it is well-formed.

15.7.4. Example

A pipeline might attempt to process a document by dispatching it to some web service. If the web service succeeds, then those results are passed to the rest of the pipeline. However, if the web service cannot be contacted or reports an error, the `p:catch` step can provide some sort of default for the rest of the pipeline.

Example 9. An Example Try/Catch

```
<p:try>
  <p:http-request method="post" href="http://example.com/form-action">
    <p:with-input>
      <p:inline content-type="application/x-www-form-urlencoded"
                >name=W3C&amp;spec=XProc</p:inline>
    </p:with-input>
  </p:http-request>
  <p:catch>
    <p:identity>
      <p:with-input port="source">
    <p:inline>
      <c:error>HTTP Request Failed</c:error>
    </p:inline>
      </p:with-input>
    </p:identity>
  </p:catch>
</p:try>
```

15.8. Atomic Steps

[Definition: An *atomic step* is a step that does not contain a subpipline when it is invoked.] The built-in steps described in [Steps 3.0] are atomic. Steps like `p:for-each` and `p:try` that always have a subpipline are *not* atomic.

Steps declared with `p:declare-step` are atomic *when they are invoked*. It is *implementation-dependent* whether or not atomic steps can be defined through some other means.

The following table gives an overview over the types of atomic steps and the terms associated with these types:

Provider	Namespace (Prefix)	Implemented in	Term
XProc processor	`http://www.w3.org/ns/xproc` (`p:`)	XProc	Processor-provided or standard (atomic) step implemented in XProc (there are currently no such steps defined)
		Other technology	Processor-provided or standard (atomic) step
	Other namespace	XProc	Processor-provided extension (atomic) step implemented in XProc
		Other technology	Processor-provided extension (atomic) step
Other (pipeline author or third party)	Other namespace	XProc	User-defined (atomic) (extension) step
		Other technology	(Third-party) (atomic) extension step

15.8.1. Processor-provided standard atomic steps

In addition to the six step types described in the preceding sections, XProc provides a standard library of atomic step types. The full vocabulary of standards steps is described in [Steps 3.0].

In addition to these standard atomic steps, other specifications by the same standardization body may define other optional steps in the XProc namespace, for example, validation or file system related steps.

Whether all steps in the XProc namespace are referred to as “standard (atomic) steps” or only the steps in the standard step library, depends on context and is intentionally

kept fuzzy. Steps in the XProc namespace that are not included in the standard step library may also be referred to as "optional standard steps" if further distinction is required.

All of the standard (including optional), atomic steps are invoked in the same way:

```
<p:atomic-step
  name? = NCName>
    (p:with-input |
     p:with-option)*
</p:atomic-step>
```

Where "*p:atomic-step*" **must** be in the XProc namespace and **must** either be declared in the standard library or in an optional standard library for the XProc version supported by the processor (see Section 13, "Versioning Considerations").

Like the aforementioned processor-provided steps, hypothetical processor-provided atomic steps *implemented in XProc* are also in the XProc namespace and need not be explicitly imported by the surrounding pipeline.

15.8.2. Extension Steps

Pipeline authors may also have access to additional steps not defined or described by this specification. Such an *external step* is invoked just like the standard steps:

```
<ext:atomic-step
  name? = NCName>
    (p:with-input |
     p:with-option)*
</ext:atomic-step>
```

Extension steps **must not** be in the XProc namespace and there **must** be a *visible* step declaration at the point of use (see Section 14.2, "Scoping of Names").

If the relevant step declaration has no *subpipeline*, then that step invokes the declared *external step*, which the processor must know how to perform. These steps are implementation-defined extensions.

If the relevant step declaration has a *subpipeline*, then that step runs the declared subpipeline. These steps are user- or implementation-defined extensions. Pipelines

can refer to themselves (recursion is allowed), to pipelines defined in imported libraries, and to other pipelines in the same library if they are in a library.

It is a *static error* (`err:XS0010`) if a pipeline contains a step whose specified inputs, outputs, and options do not *match* the *signature* for steps of that type.

It is a *dynamic error* (`err:XD0017`) if the running pipeline attempts to invoke an *external step* which the processor does not know how to perform.

The presence of other *compound steps* is *implementation-defined*; XProc provides no standard mechanism for defining them or describing what they can contain. It is a *static error* (`err:XS0048`) to use a declared step as a *compound step*.

16. Other pipeline elements

16.1. p:input

The declaration of an input identifies the name of the port, whether or not the port accepts a sequence, whether or not the port is a *primary input port*, what content types it accepts, and may provide a connection to default inputs for the port.

An input *declaration* has the following form:

```
<p:input
  port = NCName
  sequence? = boolean
  primary? = boolean
  select? = XPathExpression
  content-types? = ContentTypes
  href? = { anyURI }
  exclude-inline-prefixes? = ExcludeInlinePrefixes>
    ((p:empty |
       (p:document |
        p:inline)*) |
     anyElement*)
</p:input>
```

The attributes that can appear on `p:input` are the common attributes and:

port
The port attribute defines the name of the port. It is a *static error* (err:XS0011) to identify two ports with the same name on the same step.

sequence
The sequence attribute determines whether or not a sequence of documents is allowed on the port. If sequence is not specified, or has the value false, then it is a *dynamic error* (err:XD0006) unless exactly one document appears on the declared port.

primary
The primary attribute is used to identify the *primary input port*. An input port is a *primary input port* if primary is specified with the value true or if the step has only a single input port and primary is not specified. It is a *static error* (err:XS0030) to specify that more than one input port is the primary.

select
If a connection is provided in the declaration, then select may be used to select a portion of the input identified by the p:empty, p:document, or p:inline elements in the p:input.

With the exception that p:input cannot establish a connection to another step, what is said about the select attribute in p:with-input equally applies to p:input.

The select expression on p:input applies *only* if the default connection is used. If an explicit connection is provided by the caller, then the default select expression is ignored.

content-types
The content-types attribute lists one or more (space separated) content types that this input port will accept. If the attribute is not specified, */* is assumed. See Section 3.4, "Specifying content types".

href
As described in p:with-input.

exclude-inline-prefixes
The exclude-inline-prefixes allows the pipeline author to exclude some namespace declarations in inline content, see p:inline.

On p:declare-step, any binding provided in p:input is a default connection for the port, if no other connection is provided, see Section 16.2.1, "Connection precedence".

The p:pipe element is explicitly excluded from a declaration because it would make the default value of an input dependent on the execution of some part of the pipeline. If a runtime binding is provided for an input port, implementations **must not** attempt to dereference the default bindings. In the case of p:document connections, the URI is dereferenced only when the default connection is actually used, not during static analysis.

16.2. p:with-input

An input *connection* has the following form:

```
<p:with-input
  port? = NCName
  select? = XPathExpression
  href? = { anyURI }
  pipe? = string
  exclude-inline-prefixes? = ExcludeInlinePrefixes>
    ((p:empty |
       (p:document |
        p:pipe |
        p:inline)*) |
     anyElement*)
</p:with-input>
```

The attributes that can appear on p:with-input are the common attributes and:

port
: If the port is specified, then this is a binding for the specified port. If no port is specified, then:

  - In a p:viewport or p:for-each, it is a binding for the step's single, *anonymous input* port.
  - In a p:choose, p:when or p:if, it is a binding for the context item for the test expression(s).

- Elsewhere, it is a binding for the primary input port of the step in which it occurs. It is a *static error* (`err:XS0065`) if there is no primary input port.

It is a *static error* (`err:XS0043`) to specify a port name on `p:with-input` for `p:for-each`, `p:viewport`, `p:choose`, `p:when`, or `p:if`.

It is a *static error* (`err:XS0114`) if a port name is specified and the step type being invoked does not have an input port declared with that name.

It is a *static error* (`err:XS0086`) to provide more than one `p:with-input` for the same port.

If no connection is provided for a *primary input port*, the input will be connected to the *default readable port*. It is a *static error* (`err:XS0032`) if no connection is provided and the *default readable port* is undefined.

`select`

If a `select` expression is specified, it is effectively a filter on the input. The expression will be evaluated once for each document that appears on the port, using that document as the context item. The result of evaluating the expression (on each document that appears, in the order they arrive) will be the sequence of items that the step receives on the port.

The rules as stated in Section 3.3, "Creating documents from XDM step results" will be applied to the members of this sequence and will turn these into separate documents. It is a *dynamic error* (`err:XD0016`) if the select expression on a `p:input` or `p:with-input` returns attribute nodes or function items.

If no documents are received, the expression is not evaluated and the step receives no input on the port.

For example:

```
<p:with-input port="source">
  <p:document href="http://example.org/input.html"/>
</p:with-input>
```

provides a single document, but

```
<p:with-input xmlns:html="http://www.w3.org/1999/xhtml"
              port="source" select="//html:div">
  <p:document href="http://example.org/input.html"/>
</p:with-input>
```

provides a sequence of zero or more documents, one for each `html:div` in `http://example.org/input.html`. (Note that in the case of nested `html:div` elements, this will result in the same content being returned in several documents.)

A select expression can equally be applied to input read from another step. This input:

```
<p:with-input xmlns:html="http://www.w3.org/1999/xhtml"
              port="source" select="//html:div">
  <p:pipe step="origin" port="result"/>
</p:with-input>
```

provides a sequence of zero or more documents, one for each `html:div` in the document (or each of the documents) that is read from the `result` port of the step named `origin`.

The base URI of the document that results from a select expression is the base URI of the matched element or document. The document does not have a base URI if it results from selecting an atomic value.

`href`

The `href` attribute is a shortcut for a `p:document` child with an `href` attribute having the same value as this `href` attribute.

If `href` is specified, it is a *static error* (`err:XS0081`) if any child elements other than `p:documentation` and `p:pipeinfo` are present.

It is a *static error* (`err:XS0085`) if both a `href` attribute and a `pipe` attribute are present.

`pipe`

The `pipe` attribute is a shortcut for one or more `p:pipe` children. The attribute value **must** be whitespace-separated list of tokens or empty. It is a *static error* (`err:XS0090`) if the value of the `pipe` attribute contains any tokens not of

the form *port-name*, *port-name*`@`*step-name*, or `@`*step-name*. If "*port-name*" is omitted, the connection is to the primary output port of the step named "*step-name*". If "`@`*step-name*" is omitted, the connection is to the specified port on the same step as the step associated with the default readable port. If the value is empty, the connection is to the default readable port.

If `pipe` is specified, it is a *static error* (`err:XS0082`) any child elements other than `p:documentation` and `p:pipeinfo` are present.

It is a *static error* (`err:XS0085`) if both an `href` attribute and a `pipe` attribute are present.

`exclude-inline-prefixes`
: The `exclude-inline-prefixes` allows the pipeline author to exclude some namespace declarations in inline content, see `p:inline`.

A `p:with-input` element with no children (*e.g.*, "`<p:with-input/>`") is treated implicitly as if it contained only "`<p:pipe/>`", which is in turn equivalent to a binding to the default readable port.

If the `p:with-input` contains elements not in the XProc namespace, they are implicit inlines.

16.2.1. Connection precedence

XProc 3.0 introduces a number of new connection defaulting mechanisms to make pipeline authoring easier. Defaults only apply if there's no explicit connection, and they apply differently to primary and secondary inputs.

Primary input ports
: For a given primary input port:

1. If there is a `p:with-input` for that port and it provides a binding, even an implicit one, that binding is used.
2. If there's no `p:with-input` for that port and there is a default readable port, the input will be connected to the default readable port.

3. If there's no p:with-input for that port and there's no default readable port, then the default connection from the declaration's p:input will be used. It will be a *static error* (err:XS0032) if there is no default connection.

Secondary input ports

For a given secondary input port:

1. If there is a p:with-input for that port and it provides a binding, even an implicit one, that binding is used.

2. If there's no p:with-input for that port then the default connection from the declaration's p:input will be used. It will be a *static error* (err:XS0032) if there is no default connection.

16.3. p:output

A p:output identifies an output port.

```
<p:output
  port? = NCName
  sequence? = boolean
  primary? = boolean
  content-types? = ContentTypes />
```

The attributes that can appear on p:output are the common attributes and:

port

The port attribute defines the name of the port. It is a *static error* (err:XS0011) to identify two ports with the same name on the same step.

sequence

An output declaration can indicate if a sequence of documents is allowed to appear on the declared port. If sequence is specified with the value true, then a sequence is allowed. If sequence is not specified on p:output, or has the value false, then it is a *dynamic error* (err:XD0007) if the step does not produce exactly one document on the declared port.

primary

The primary attribute is used to identify the primary output port. An output port is a primary output port if primary is specified with the value true or if the step has only a single output port and primary is not specified. It is a *static error* (err:XS0014) to identify more than one output port as primary.

content-types

An output declaration can indicate the content types of the documents appearing on that port. If content-types is specified then only documents matching these content types are allowed to appear on that port. If the attribute is not specified, */* is assumed. It is a *dynamic error* (err:XD0042) if a document arrives on an output port whose content type is not accepted by the output port specification.

> **Note**
>
> Implementations are free to perform static checking of the connected ports and indicate that the content types of the connected ports will not match, however they **must not** raise an error statically.

On *compound steps*, the declaration **may** be accompanied by a *connection* for the output.

```
<p:output
  port? = NCName
  sequence? = boolean
  primary? = boolean
  content-types? = ContentTypes
  href? = { anyURI }
  pipe? = string
  exclude-inline-prefixes? = ExcludeInlinePrefixes>
    ((p:empty |
       (p:document |
        p:pipe |
        p:inline)*) |
     anyElement*)
</p:output>
```

The additional attributes that can appear on an output declaration on a compound step are:

`href`
: As described in `p:with-input`.

`pipe`
: As described in `p:with-input`.

`exclude-inline-prefixes`
: The `exclude-inline-prefixes` allows the pipeline author to exclude some namespace declarations in inline content, see `p:inline`.

Finally, on a `p:declare-step` that declares a pipeline, the `p:output` can specify serialization options.

```
<p:output
  port? = NCName
  sequence? = boolean
  primary? = boolean
  content-types? = ContentTypes
  href? = { anyURI }
  pipe? = string
  exclude-inline-prefixes? = ExcludeInlinePrefixes
  serialization? = map(xs:QName,item()*)>
    ((p:empty |
       (p:document |
        p:pipe |
        p:inline)*) |
     anyElement*)
</p:output>
```

`serialization`
: The `serialization` attribute can be used to provide serialization parameters.

It is a *static error* (`err:XS0029`) to specify a connection for a `p:output` inside a `p:declare-step` for an *external step*.

If a connection is provided for a `p:output`, documents are *read from* that connection and those documents form the output that *is written* to the output port. In other words, placing a `p:document` inside a `p:output` causes the processor to *read that*

document and provide it on the output port. It *does not* cause the processor to *write* the output to that document.

16.3.1. Serialization parameters

The `serialization` attribute allows the user to request serialization parameters on an output port. These parameters control serialization as defined by [Serialization].

If the pipeline processor serializes the output on a port, it **must** use the serialization parameters specified. If a `serialization` document property is present, the serialization properties specified by the `serialization` document property must be merged with the properties specified with the `serialization` attribute first. For further details see the explanation of the `serialization` document property in Section 3.1, "Document Properties".

If the processor is not serializing (if, for example, the pipeline has been called from another pipeline), then serialization does not apply. The serialization parameter map is computed (and must therefore be statically and syntactically valid), but the processor **must not** raise an error if the output could not be serialized with those parameters.

It is a *dynamic error* (`err:XD0020`) if the combination of serialization options specified or defaulted is not allowed. Implementations **must** check that all of the specified serialization options are allowed if they serialize the specified output. If the specified output is not being serialized implementations **may** but are not required to check that the specified options are allowed.

In order to be consistent with the rest of this specification, values for boolean serialization parameters can also use one of the XML Schema lexical forms for boolean: `true`, `false`, `1`, or `0`. This is different from the [Serialization] specification, which uses `yes` and `no`. No change in semantics is implied by this different spelling.

The default value of any serialization parameters not specified on a particular output is *implementation-defined*.

16.3.1.1. Serialization method

The `method` option controls the serialization method used by this component with standard values of `html`, `xml`, `xhtml`, `text` and `json`. Only the `xml` value is required to be supported. Implementations may support other method values but their results are *implementation-defined*.

If the serialization parameter `method` is not specified, the processor **should** select a method based on the document's `content-type` property:

- For documents with content types `application/xml`, `text/xml`, and `application/*+xml` (except for `application/xhtml+xml`), serialization method `xml` should be used.
- For documents with content type `application/xhtml+xml` serialization method `xhtml` should be used.
- For documents with content type `text/html` serialization method `html` should be used.
- For documents with *text media types* serialization method `text` should be used.
- For documents with *JSON media types* serialization method `json` should be used.
- The serialization method for documents with other media types is *implementation-defined*.

If serialization method `xml` or `html` (if supported) is chosen, either explicitly or implicitly, the following default values **must** be used:

- Parameter `version` is set to `1.0`.
- Parameter `encoding` is set to `UTF-8`.
- Parameter `omit-xml-declaration` is set to `false`.

16.3.1.2. Minimal conformance

A minimally conforming implementation must support the `xml` output method with the following option values:

- The `version` must support the value `1.0`.
- The `encoding` must support the value `UTF-8`.
- The `omit-xml-declaration` must be supported.

All other option values may be ignored for the `xml` output method.

If a processor chooses to implement an option for serialization, it **must** conform to the semantics defined in the [Serialization] specification.

16.4. Variables and Options

Variables and options provide a mechanism for pipeline authors to construct temporary results and hold onto them for reuse.

Variables are created in compound steps and, like XSLT variables, are single assignment, though they may be shadowed by subsequent declarations of other variables with the same name.

Options can be declared on atomic or compound steps. The value of an option can be specified by the caller invoking the step. Any value specified by the caller takes precedence over the default value of the option.

16.4.1. p:variable

A `p:variable` declares a variable and associates a value with it. Variable declarations may optionally specify the type of the variable using an [XPath 3.1] sequence Type.

```
<p:variable
  name = EQName
  as? = XPathSequenceType
  select = XPathExpression
  collection? = boolean
  href? = { anyURI }
  pipe? = string
  exclude-inline-prefixes? = ExcludeInlinePrefixes>
    ((p:empty |
       (p:document |
        p:pipe |
        p:inline)*) |
     anyElement*)
</p:variable>
```

The attributes that can appear on `p:variable` are the common attributes and:

`name`

The name of the variable **must** be an EQName. If it does not contain a prefix then it is in no namespace. It is a *static error* (`err:XS0028`) to declare an option or variable in the XProc namespace. It is a *static error* (`err:XS0087`) if the name attribute on `p:option` or `p:variable` has a prefix which is not bound to a namespace.

It is a *static error* (`err:XS0088`) if the qualified name of a `p:variable` *shadows* the name of a static option.

`as`

The type of the value may be specified in the `as` attribute using an XPath sequence type, see Section 11.4, "Variable and option types".

`select`

The variable's value is specified with a `select` attribute. The `select` attribute **must** be specified. The content of the `select` attribute is an XPath expression which will be evaluated to provide the value of the variable. It is a *static error* (`err:XS0094`) if a `p:variable` does not have a select attribute.

The `select` expression is evaluated as an XPath expression using the appropriate context as described in Section 7.2.2, "XPath in XProc", for the enclosing *container*. The precise details about what XPath expressions are

allowed (for example, can the expression declare a function) is *implementation-defined*.

collection
If `collection` is unspecified or has the value `false`, then it has no effect.

If `collection` is `true`, the context item is undefined. All of the documents that appear on the connection for the `p:variable` will be available as the default collection within `select` expression.

href
As described in `p:with-input`.

pipe
As described in `p:with-input`.

exclude-inline-prefixes
The `exclude-inline-prefixes` allows the pipeline author to exclude some namespace declarations in inline content, see `p:inline`.

Steps are connected together by their input and output ports. Variables are connected to steps by their input, which provides the context node for the expression, and by the expressions that contain references to them. Any step which contains a reference to a variable effectively consumes the "output" of the variable. It is a *static error* (`err:XS0076`) if there are any loops in the connections between steps and variables: no step can refer to a variable if there is any sequence of connections from that step that leads back to the input that provides the context node for the expression that defines the value of the variable.

If `collection` is true, the context item for the expression is undefined. Otherwise, the context item for the expression comes from the document connections, if they are specified. If they are not specified, the context item comes from the *default readable port* (computed as if `p:variable` was an atomic step). If no *default readable port* exists, the context item is undefined.

It is a *dynamic error* (`err:XD0001`) if an XPath expression makes reference to the context item, size, or position when the context item is undefined. It is a *dynamic error* (`err:XD0065`) to refer to the context item, size, or position if a sequence of documents appears on the connection that provides the context.

Since all *in-scope bindings* are present in the Processor XPath Context as variable bindings, `select` expressions may refer to the value of *in-scope bindings* by variable reference.

16.4.2. p:option

A `p:option` declares an option and associates a default value with it. Option declarations may optionally specify the type of the option using an [XPath 3.1] sequence Type.

```
<p:option
  name = EQName
  as? = XPathSequenceType
  values? = string
  static? = boolean
  required? = boolean
  select? = XPathExpression
  visibility? = private|public />
```

The attributes that can appear on `p:option` are the common attributes and:

`name`

The name of the option **must** be an EQName. If it does not contain a prefix then it is in no namespace. It is a *static error* (`err:XS0028`) to declare an option or variable in the XProc namespace. It is a *static error* (`err:XS0087`) if the name attribute on `p:option` or `p:variable` has a prefix which is not bound to a namespace.

It is a *static error* (`err:XS0088`) if the qualified name of a `p:option` *shadows* the name of a static option.

`as`

The type of the value may be specified in the `as` attribute using an XPath sequence type, see Section 11.4, "Variable and option types".

`values`

A list of acceptable values may be specified in the `values` attribute. If specified, the value of the `values` attribute **must** be a list of atomic values expressed as an

XPath sequence, for example: `('one', 'two', 'three')`. It is a *static error* (`err:XS0101`) if the values list is not an XPath sequence of atomic values.

The values list is an additional constraint on the acceptable values for the option. It is a *dynamic error* (`err:XD0019`) if an option declares a list of acceptable values and an attempt is made to specify a value that is not a member of that list.

The option value must satisfy the `as` type, if one is provided, and must be equal to (XPath "`eq`") one of the listed `values`. It is possible to combine `as` and `values` in ways that exclude all actual values (for example, `as="xs:integer"` and `values="(1.5,'pi')"`). Doing so will make it impossible to specify a value for the option.

`static`
An indication of whether the option is to be evaluated statically or not. See Section 11.3, "Static Options". If `static` is not specified, it defaults to "`false`".

`required`
An option may declare that it is required by specifying the value `true` for the `required` attribute. If an option is required, it is a *static error* (`err:XS0018`) to invoke the step without specifying a value for that option. If `required` is not specified, it defaults to "`false`".

`select`
If an option is not required, its default value may be specified with a `select` attribute. If no default value is specified, the default value is the empty sequence.

If specified, the content of the `select` attribute is an XPath expression which will be evaluated to provide the default value for the option.

The default value of an option is specified with an XPath expression. It must be a statically valid expression at that point. Consequently, if it contains option references, these can only be references to preceding non-static options on the step or to in-scope static options. It is a *dynamic error* (`err:XD0001`) if an XPath expression makes reference to the context item, size, or position when the context item is undefined. This error will always arise if the `select` expression refers to the context item because there can never be a context item for `p:option` default values.

The precise details about what XPath expressions are allowed (for example, can the expression declare a function) is *implementation-defined*.

`visibility`
: If the `p:option` is a child of a `p:library`, the `visibility` attribute controls whether the option is visible to an importing pipeline. If `visibility` is set to "`private`", the option is visible inside the `p:library` but not visible to any pipeline importing the `p:library`. If the visibility attribute is missing, "`public`" is assumed. If the `p:option` is not a child of `a p:library` the attribute has no effect and is ignored.

It is a *static error* (`err:XS0004`) to declare two or more options on the same step with the same name.

The following errors apply to options:

- It is a *static error* (`err:XS0017`) to specify that an option is both `required` *and* has a default value.
- It is a *static error* (`err:XS0095`) to specify that an option is both `required` *and* static.

The pipeline author may use `p:with-option` on a step when it is invoked. Values specified with `p:with-option` override any default values specified.

16.4.3. p:with-option

A `p:with-option` provides an actual value for an option when a step is invoked.

```
<p:with-option
  name = EQName
  as? = XPathSequenceType
  select = XPathExpression
  collection? = boolean
  href? = { anyURI }
  pipe? = string
  exclude-inline-prefixes? = ExcludeInlinePrefixes>
    ((p:empty |
       (p:document |
        p:pipe |
        p:inline)*) |
     anyElement*)
</p:with-option>
```

The attributes that can appear on `p:with-option` are the common attributes and:

`name`

The name of the option **must** be a EQName. If it does not contain a prefix then it is in no namespace. It is a *static error* (`err:XS0031`) to use an option name in `p:with-option` if the step type being invoked has not declared an option with that name.

It is a *static error* (`err:XS0080`) to include more than one `p:with-option` with the same option name as part of the same step invocation.

`as`

The type of the value may be specified in the `as` attribute using an XPath sequence type, see Section 11.4, "Variable and option types".

`select`

The actual value is specified with a `select` attribute. The `select` attribute **must** be specified. The value of the `select` attribute is an XPath expression which will be evaluated to provide the value of the option.

`collection`

If `collection` is unspecified or has the value `false`, then it has no effect.

If `collection` is `true`, the context item is undefined. All of the documents that appear on the connection for the `p:with-option` will be available as the default collection within `select` expression.

href
: As described in p:with-input.

pipe
: As described in p:with-input.

exclude-inline-prefixes
: The exclude-inline-prefixes allows the pipeline author to exclude some namespace declarations in inline content, see p:inline.

Any p:with-option which contains a reference to a variable effectively consumes the "output" of the p:variable or p:option that defines that variable. It is a *static error* (err:XS0076) if there are any loops in the connections between steps and variables: no step can refer to a variable if there is any sequence of connections from that step that leads back to the input that provides the context node for the expression that defines the value of the variable.

If collection is true, the context item for the expression is undefined. Otherwise, the context item for the expression comes from the document connections, if they are specified. If they are not specified, the context item comes from the *default readable port* of the step. If no *default readable port* exists, the context item is undefined.

It is a *dynamic error* (err:XD0001) if an XPath expression makes reference to the context item, size, or position when the context item is undefined. It is a *dynamic error* (err:XD0065) to refer to the context item, size, or position if a sequence of documents appears on the connection that provides the context.

Since all *in-scope bindings* are present in the Processor XPath Context as variable bindings, select expressions may refer to the value of *in-scope bindings* by variable reference.

It is a *static error* (err:XS0092) if a p:with-option attempts to change the value of an option that is declared static. See Section 11.3, "Static Options".

16.4.3.1. Syntactic Shortcut for Option Values

Namespace qualified attributes on a step are *extension attributes*. Attributes, other than name, that are not namespace qualified are treated as a syntactic shortcut for

specifying the value of an option. In other words, the following two steps are equivalent:

The first step uses the standard `p:with-option` syntax:

```
<ex:stepType>
  <p:with-option name="option-name" select="'some value'"/>
</ex:stepType>
```

The second step uses the syntactic shortcut:

```
<ex:stepType option-name="some value"/>
```

There are some limitations to this shortcut syntax:

1. It only applies to option names that are not in a namespace.
2. It only applies to option names that are not otherwise used on the step, such as "`name`".

For the value of an option's syntactic shortcut attribute, the following applies:

- [Definition: A *map attribute* is an option's syntactic shortcut attribute for which the option's sequence type is a map or array.] The attribute's value is interpreted directly as an XPath expression, which must result in a value of the applicable datatype.
- For any other option's sequence type it is considered an *attribute value template*. The context node for the attribute value template comes from the default readable port for the step on which they occur. If there is no such port, the context node is undefined.

  As with other attribute value templates, the attribute's string value, as an `xs:untypedAtomic`, is used as the value of the option. Function conversion rules apply to convert this untyped atomic value to the option's sequence type.

It is a *static error* (`err:XS0027`) if an option is specified with both the shortcut form and the long form. It is a *static error* (`err:XS0031`) to use an option on an *atomic step*

that is not declared on steps of that type. It is a *static error* (`err:XS0092`) to specify a value for an option that is declared static.

The syntactic shortcuts apply equally to standard atomic steps and extension atomic steps.

16.5. p:declare-step

A `p:declare-step` provides the type and *signature* of a pipeline or an *external step*. Pipelines contain a subpipeline which defines what the declared step does. [Definition: An *external step* is one supported by the implementation, but which has no exposed subpipeline.]

The standard XProc atomic steps (`p:add-attribute`, `p:add-xml-base` …) are all external steps. Whether or not an implementation allows users to provide their own external steps is *implementation-dependent*. A `p:declare-step` must be provided for every pipeline and external step that is used in a pipeline.

When a declared step is evaluated directly by the XProc processor (as opposed to occurring as an atomic step in some *container*), how the input and output ports are connected to documents is *implementation-defined*.

A step declaration is not a step in its own right. Sibling steps cannot refer to the inputs or outputs of a `p:declare-step` using `p:pipe`; only instances of the type can be referenced.

16.5.1. Declaring pipelines

When a `p:declare-step` declares a pipeline, that pipeline encapsulates the behavior of the specified *subpipeline*. Its children declare inputs, outputs, and options that the pipeline exposes and identify the steps in its subpipeline.

```
<p:declare-step
  name? = NCName
  type? = EQName
  psvi-required? = boolean
  xpath-version? = decimal
  exclude-inline-prefixes? = ExcludeInlinePrefixes
  version? = 3.0
  visibility? = private|public>
    (p:import |
     p:import-functions)*,
    (p:input |
     p:output |
     p:option)*,
    p:declare-step*,
    subpipeline?
</p:declare-step>
```

The attributes that can appear on `p:declare-step` are the common attributes and:

`name`

The `name` attribute provides a name for the step. This name can be used within the subpipeline to refer back to the declaration, for example, to read from its inputs. See also Section 2.1.1, "Step names".

`type`

The `type` attribute provides a type for the step. Step types are used as the name of the element by which the step is invoked. See also Section 2.1.2, "Step types".

The value of the `type` can be from any namespace provided that the expanded-QName of the value has a non-null namespace URI. It is a *static error* (`err:XS0025`) if the expanded-QName value of the `type` attribute is in no namespace or in the XProc namespace. Neither users nor implementers may define additional steps in the XProc namespace.

`psvi-required`

The `psvi-required` attribute allows the author to declare that a step relies on the processor's ability to pass PSVI annotations between steps, see Section 9, "PSVIs in XProc". If the attribute is not specified, the value "`false`" is assumed.

`xpath-version`

The requested `xpath-version` **must** be used to evaluate XPath expressions subject to the constraints outlined in Section 7.2.2, "XPath in XProc". If the

attribute is not specified, the value "`3.1`" is assumed. It is a *static error* (`err:XS0110`) if the requested XPath version is less than "`3.1`".

`exclude-inline-prefixes`
The a description of `exclude-inline-prefixes`, see `p:inline`.

`version`
The `version` attribute identifies the version of XProc for which this step declaration was authored. If the `p:declare-step` has no ancestors in the XProc namespace, then it **must** have a `version` attribute. It is a *static error* (`err:XS0062`) if a required version attribute is not present. See Section 13, "Versioning Considerations".

`visibility`
If the `p:declare-step` is a child of a `p:library` the `visibility` attribute controls whether the step is visible to an importing pipeline. If `visibility` is set to `private`, the step type is only visible inside the `p:library` and is not visible to any pipeline importing the `p:library`. If the `visibility` attribute is missing, `public` is assumed. If the `p:declare-step` is not a child of a `p:library` the attribute has no effect and is ignored.

In the general case, the children of a `p:declare-step` can be grouped into several sections. All of these sections, except the subpipeline, may be empty.

1. Imports must come first.
2. The prologue follows the imports. [Definition: The *prologue* consists of the `p:input`, `p:output`, and `p:option` elements.]
3. The prologue may be followed by any number of inline `p:declare-step` elements that declare additional steps.
4. Finally, there must be at least one step in the subpipeline.

Options in the prologue may not shadow each other. It is a *static error* (`err:XS0091`) if an `p:option` shadows another option declared within the same `p:declare-step`. (Within the subpipeline, variables may shadow (non-static) options and lexically preceding variables.)

The prologue ends with additional `p:declare-step` elements, if any, and is followed by the subpipeline. Any step imported or declared in the prologue of a pipeline may be invoked as a step within the subpipeline of that pipeline.

The environment inherited by the *subpipeline* is the *empty environment* with these modifications:

- All of the declared inputs are added to the *readable ports* in the environment.
- If a *primary input port* is declared, that port is the *default readable port*, otherwise the default readable port is undefined.
- The *in-scope bindings* at the beginning of a `p:declare-step` are limited to the lexically preceding, statically declared options.

If a *primary output port* is declared and that port has no *connection*, then it is connected to the *primary output port* of the *last step* in the *subpipeline*. It is a *static error* (`err:XS0006`) if the primary output port is unconnected and the *last step* in the subpipeline does not have a primary output port.

16.5.2. Declaring external steps

The distinction between a pipeline declaration and an external step declaration hinges on the presence or absence of a subpipeline. A step declaration that does not contain a subpipeline is, by definition, declaring an *external step*.

External step declarations may not import other pipelines or functions, may not declare static options, and may not declare additional steps. In other words, the content of an external step declaration consists exclusively of `p:input`, `p:output`, and `p:option` elements.

```
<p:declare-step
  name? = NCName
  type? = EQName
  psvi-required? = boolean
  xpath-version? = decimal
  exclude-inline-prefixes? = ExcludeInlinePrefixes
  version? = 3.0
  visibility? = private|public>
    (p:input |
     p:output |
     p:option)*
</p:declare-step>
```

Implementations may use *extension attributes* to provide *implementation-dependent* information about a declared step. For example, such an attribute might identify the code which implements steps of this type.

It is not an error for a pipeline to include declarations for steps that a particular processor does not know how to implement. It is, of course, an error to attempt to evaluate such steps. The function `p:step-available` will return `false` when called with the type name of such a step.

It is a *dynamic error* (`err:XD0017`) if the running pipeline attempts to invoke an external step which the processor does not know how to perform.

16.6. p:library

A `p:library` is a collection of static options, and step declarations.

```
<p:library
  psvi-required? = boolean
  xpath-version? = decimal
  exclude-inline-prefixes? = ExcludeInlinePrefixes
  version? = 3.0>
    (p:import |
     p:import-functions)*,
    p:option*,
    p:declare-step*
</p:library>
```

The `version` attribute identifies the version of XProc for which this library was authored. If the `p:library` has no ancestors in the XProc namespace, then it **must** have a `version` attribute. See Section 13, "Versioning Considerations".

The requested `xpath-version` **must** be used to evaluate XPath expressions subject to the constraints outlined in Section 7.2.2, "XPath in XProc". If the attribute is not specified, the value "`3.1`" is assumed. It is a *static error* (`err:XS0110`) if the requested XPath version is less than "`3.1`".

The `psvi-required` attribute allows the author to declare that a step relies on the processor's ability to pass PSVI annotations between steps, see Section 9, "PSVIs in XProc". If the attribute is not specified, the value "`false`" is assumed.

For a description of `psvi-required`, see Section 9, "PSVIs in XProc"; for `xpath-version`, see Section 7.2.2, "XPath in XProc"; for `exclude-inline-prefixes`, see `p:inline`.

> **Note**
>
> The steps declared in a pipeline library are referred to by their type. It is not an error to put a `p:declare-step` without a `type` in a `p:library`, but there is no standard mechanism for instantiating it or referring to it. It is effectively invisible.

Like `p:declare-step`, within a library, imports must precede the prologue (any static options), which must precede any declared steps.

Libraries can import pipelines and/or other libraries. See also Appendix H, Handling Circular and Re-entrant Library Imports (Non-Normative).

16.7. p:import

An `p:import` loads a pipeline or pipeline library, making it available in the pipeline or library which contains the `p:import`.

```
<p:import
  href = anyURI />
```

An import statement loads the specified IRI and makes any pipelines declared within it available to the current pipeline.

It is a *static error* (`err:XS0052`) if the URI of a `p:import` cannot be retrieved or if, once retrieved, it does not point to a `p:library` or `p:declare-step`. It is a *static error* (`err:XS0053`) to import a single pipeline if that pipeline does not have a `type`.

Attempts to retrieve the library identified by the URI value may be redirected at the parser level (for example, in an entity resolver) or below (at the protocol level, for example, via an HTTP Location: header). In the absence of additional information outside the scope of this specification within the resource, the base URI of the library is always the URI of the actual resource returned. In other words, it is the URI of the resource retrieved after all redirection has occurred.

As imports are processed, a processor may encounter new `p:import` elements whose library URI is the same as one it has already processed in some other context. This may happen as a consequence of resolving the URI. If the actual base URI is the same as one that has already been processed, the implementation must recognize it as the same library and should not need to process the resource. Also, a duplicate, circular chain of imports, or a re-entrant import is not an error and implementations must take the necessary steps to avoid infinite loops and/or incorrect notification of duplicate step definitions. It is not an error for a library to import itself. An example of such steps is listed in Appendix H, Handling Circular and Re-entrant Library Imports (Non-Normative).

A library is considered the same library if the base URI of the resource retrieved is the same. If different base URIs resolve to the same library (for instance when a web server returns the same document on different URLs), they must *not* be considered the same imported library.

16.8. p:import-functions

An `p:import-functions` element identifies a library of externally defined functions to be imported into the pipeline. After the functions have been imported, they are available in the processor XPath context.

```
<p:import-functions
  href = anyURI
  content-type? = ContentType
  namespace? = string />
```

href

The `href` attribute identifies the URI of the function library. It is a *static error* (`err:XS0103`) if the URI of a `p:import-functions` element cannot be retrieved or if, once retrieved, it points to a library that the processor cannot import.

content-type

The `content-type` specifies what kind of library is expected at the URI. If no type is specified, the way that the processor determines the type of the library is *implementation-defined*.

namespace

If a `namespace` is specified, it must be a whitespace separated list of namespace URIs. Only functions in those namespaces will be loaded.

The ability to import functions is optional. Whether or not a processor can import functions, and if it can, what kinds of function libraries it can import from is *implementation-defined*. Pipeline authors can use `p:function-library-importable` to test whether or not a particular kind of library can be loaded.

Importing functions from a library implies loading and processing that library according to its conventions (loading imports, resolving dependencies, etc.). It is a *static error* (`err:XS0104`) if the processor cannot load the function library. This may occur because the format is unknown, because it is a version of the library that the processor does not recognize, or if it's uninterpretable for any other reason. It is a *static error* (`err:XS0106`) if the processor detects that a particular library is unloadable. This may occur if the processor is, in principle, able to load libraries of the specified format, but detects that the particuar library requested is somehow ill-formed (syntactically invalid, has unsatisfiable dependencies or circular imports, etc.).

Imported functions must be unique (they must not have the same name, namespace, and arity). It is a *static error* (`err:XS0105`) if a function imported from a library has the same name and arity as a function already imported.

16.9. p:pipe

A `p:pipe` connects an input to a port on another step.

```
<p:pipe
  step? = NCName
  port? = NCName />
```

The `p:pipe` element connects to a readable port of another step. It identifies the readable port to which it connects with the name of the step in the `step` attribute and the name of the port on that step in the `port` attribute. It is a *static error* (`err:XS0099`) if `step` or `port` are not valid instances of `NCName`.

If the `step` attribute is not specified, it defaults to the step which provides the default readable port. If the `port` attribute is not specified, it defaults to the primary output port of the step identified (explicitly or implicitly).

It is a *static error* (`err:XS0067`) if the `step` attribute is not specified, and there is no default readable port. It is a *static error* (`err:XS0068`) if the `port` attribute is not specified, and the step identified has no primary output port.

In all cases except when the `p:pipe` is within an `p:output` of a *compound step*, it is a *static error* (`err:XS0022`) if the port identified by the `p:pipe` is not in the *readable ports* of the step that contains the `p:pipe`.

A `p:pipe` that is a *connection* for an `p:output` of a *compound step* may connect to one of the readable ports of the compound step or to an output port on one of the compound step's *contained steps*. In other words, the output of a compound step can simply be a copy of one of the available inputs or it can be the output of one of its children.

When the `p:pipe` is within an `p:output` of a *compound step*, it is a *static error* (`err:XS0078`) if the port identified by the `p:pipe` is not in the *readable ports* of the compound step and is not a readable port of a contained step.

16.10. p:inline

A `p:inline` provides a document inline.

```
<p:inline
  exclude-inline-prefixes? = ExcludeInlinePrefixes
  content-type? = string
  document-properties? = map(xs:QName,item()*)
  encoding? = string>
    anyNode*
</p:inline>
```

The `content-type` attribute can be used to set the content type of the provided document; the `document-properties` attribute can be used to set the *document properties* of the provided document.

The document's content type is determined statically. If a `content-type` is specified, that is the content type. Otherwise, the content type is "`application/xml`".

It is a *dynamic error* (`err:XD0062`) if the `document-properties` map contains a `content-type` key and that key has a value that differs from the statically determined content type.

The base URI of the document is the base URI of the `p:inline` element or of the parent element in the case of an implicit inline. If `document-properties` provides a value for "`base-uri`", this value is the base URI of the document. It is a *dynamic error* (`err:XD0064`) if the base URI is not both absolute and valid according to [RFC 3986].

How the content of a `p:inline` element is interpreted depends on the document's content type and the `encoding` attribute.

It is a *dynamic error* (`err:XD0054`) if an encoding is specified and the content type is an *XML media type* or an *HTML media type*.

It is a *dynamic error* (`err:XD0055`) if the content type value specifies a character set and the `encoding` attribute is absent.

It is a *dynamic error* (`err:XD0039`) if the `encoding` attribute is present and content type value specifies a character set that is not supported by the implementation.

It is a *dynamic error* (`err:XD0056`) if an encoding is specified and the content of the `p:inline` contains any XML markup. It is a *dynamic error* (`err:XD0063`) if the `p:inline` contains any XML markup and has a content type that is not an *XML media*

type or an *HTML media type*. In other words, in these cases, the entire content must be a single text node. CDATA sections and character references do not count as markup for this purpose because they will already have been replaced by the XML parser that read the pipeline.

If the `encoding` attribute is present, the content must be decoded. The encoding value "`base64`" **must** be supported and identifies the content as being base64-encoded. An implementation may support encodings other than `base64`, but these encodings and their names are *implementation-defined*. It is a *static error* (`err:XS0069`) if the encoding specified is not supported by the implementation. It is a *dynamic error* (`err:XD0040`) if the body is not correctly encoded per the value of the `encoding` attribute.

If an `encoding` attribute is present, value templates are never expanded. The value of `[p:]expand-text` is irrelevant and always ignored. Otherwise, the text content of `p:inline` is subject to text value template expansion irrespective of its content type. (Attribute value template expansion only applies to XML and HTML media types.)

The interpretation of the (possibly decoded) content depends on the document's content type.

> **Note**
>
> In the presence of *text value templates*, it is not possible to interpret the non-XML characters until the templates have been expanded.

16.10.1. Inline XML and HTML content

If `content-type` is not specified or specifies an *XML media type* or an *HTML media type*, then the content is XML. A new XML document is created by wrapping a document node around the nodes which appear as children of `p:inline`.

The in-scope namespaces of the inline document differ from the in-scope namespace of the content of the `p:inline` element in that bindings for all its *excluded namespaces*, as defined below, are removed:

- The XProc namespace itself (`http://www.w3.org/ns/xproc`) is excluded.
- A namespace URI designated by using an `exclude-inline-prefixes` attribute on the enclosing `p:inline` is excluded.
- A namespace URI designated by using an `exclude-inline-prefixes` attribute on any ancestor `p:declare-step` or `p:library` is also excluded. (In other words, the effect of several `exclude-inline-prefixes` attributes among the ancestors of `p:inline` is cumulative.)

The value of each prefix in the `exclude-inline-prefixes` attribute is interpreted as follows:

- The value of the attribute is either `#all`, or a whitespace-separated list of tokens, each of which is either a namespace prefix or `#default`. The namespace bound to each of the prefixes is designated as an excluded namespace. It is a *static error* (`err:XS0057`) if the `exclude-inline-prefixes` attribute does not contain a list of tokens or if any of those tokens (except `#all` or `#default`) is not a prefix bound to a namespace in the in-scope namespaces of the element on which it occurs.
- The default namespace of the element on which `exclude-inline-prefixes` occurs may be designated as an excluded namespace by including `#default` in the list of namespace prefixes. It is a *static error* (`err:XS0058`) if the value `#default` is used within the `exclude-inline-prefixes` attribute and there is no default namespace in scope.
- The value `#all` indicates that all namespaces that are in scope for the element on which `exclude-inline-prefixes` occurs are designated as excluded namespaces.

The XProc processor **must** include all in-scope prefixes that are not explicitly excluded. If the namespace associated with an excluded prefix is used in the expanded-QName of a descendant element or attribute, the processor **may** include that prefix anyway, or it may generate a new prefix.

Consider this example:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                xmlns:c="http://www.w3.org/ns/xproc-step"
                version="3.0">
  <p:output port="result" serialization="map { 'indent': true() }"/>

  <p:identity xmlns:a="http://example.com/a"
              xmlns:b="http://example.com/b"
              xmlns:c="http://example.com/c">
    <p:with-input port="source">
      <p:inline exclude-inline-prefixes="a b">
        <doc>
          <b:part/>
        </doc>
      </p:inline>
    </p:with-input>
  </p:identity>

</p:declare-step>
```

which might produce a result like this:

```
        <doc xmlns:c="http://example.com/c">
           <b:part xmlns:b="http://example.com/b"/>
        </doc>
```

The declaration for "c" must be present because it was not excluded. The "part" element uses the namespace bound to "b", so *some* binding must be present. In this example, the original prefix has been preserved, but it would be equally correct if a different prefix had been used.

The text-node descendants of a p:inline may be text value templates. Attribute descendants may be attribute value templates. This is controlled by the [p:]expand-text and the p:inline-expand-text attribute. See Section 14.9.1, "Expand text attributes".

16.10.2. Inline text content

If the document's content type is a *text media type*, then the content is text. A new text document is created by joining the text nodes which appear as children of p:inline

together to a single text node and wrapping a document node around it. Any preceding or following whitespace-only text nodes will be preserved.

16.10.3. Inline JSON content

If the document's content type is a *JSON media type*, then the context is JSON. A new JSON document is created by joining the text values of children of p:inline together and parse it as JSON.

It is a *dynamic error* (`err:XD0057`) if the text content does not conform to the JSON grammar.

16.10.4. Other inline content

How a processor interprets other media types is *implementation-defined*.

16.10.5. Implicit inlines

As an authoring convenience, `p:inline` may be omitted if one or more element nodes, optionally preceded and/or followed by whitespace occurs where a `p:inline` is allowed. Whitespace around each element is ignored and the element is treated as if it was enclosed within a `p:inline` element (with no attributes). Elements in the XProc namespace are forbidden except for `p:documentation` and `p:pipeinfo` which are ignored.

The following example demonstrates this implicit behaviour:

```
<p:identity name="identity" code="my:implicitinline1">
    <p:with-input port="source">
        <p xmlns="http://example.org/ns">Text</p>
        <p xmlns="http://example.org/ns">Other text</p>
    </p:with-input>
</p:identity>
```

Is interpreted as follows:

```
<p:identity name="identity" code="my:implicitinline2">
    <p:with-input port="source">
        <p:inline><p xmlns="http://example.org/ns">Text</p></p:inline>
        <p:inline><p xmlns="http://example.org/ns">Other text</p></p:inline>
    </p:with-input>
</p:identity>
```

An explicit `p:inline` is required if the author wants to include top level comments, processing instructions, or whitespace, or if the document element is in the XProc namespace.

It is a *static error* (`err:XS0079`) if comments, non-whitespace text nodes, or processing instructions occur as siblings of an element node that would be treated as an implicit inline.

16.11. p:document

A `p:document` reads a document from a URI.

```
<p:document
  href = { anyURI }
  content-type? = string
  document-properties? = map(xs:QName,item()*)
  parameters? = map(xs:QName,item()*) />
```

The value of the `href` attribute, after expanding any *attribute value templates*, is a URI. The URI is interpreted as an IRI reference. If it is relative, it is made absolute against the base URI of the `p:document` element. It is a *dynamic error* (`err:XD0064`) if the base URI is not both absolute and valid according to [RFC 3986].

The semantics of `p:document` are the same as the semantics of `p:load` where the `href` option is the URI, the `content-type` option comes from `content-type` attribute, the `document-properties` option comes from the `document-properties` attribute, and the `parameters` option comes from the `parameters` attribute.

> **Note**
>
> A `p:document` always *reads* from the specified IRI. In the context of a `p:input` or `p:with-input`, this seems perfectly natural. In the context of a `p:output`, this may seem a little asymmetrical. Putting a `p:document` in a `p:output` causes the pipeline to *read* from the specified IRI and provide that document *as an output* on that port.
>
> Use `p:store` to store the results that appear on a `p:output`.

16.12. p:empty

A `p:empty` connects to an *empty sequence* of documents.

```
<p:empty />
```

If an empty binding is used, it must be the only binding for the port. It is a *static error* (`err:XS0089`) if the `p:empty` binding appears as a sibling of any other binding, including itself.

16.13. p:documentation

A `p:documentation` contains human-readable documentation.

```
<p:documentation>
    any-well-formed-content*
</p:documentation>
```

There are no constraints on the content of the `p:documentation` element. Documentation is ignored by pipeline processors. See Section 14.6, “Documentation”.

16.14. p:pipeinfo

A `p:pipeinfo` contains ancillary information for steps in the pipeline.

```
<p:pipeinfo>
    any-well-formed-content*
</p:pipeinfo>
```

There are no constraints on the content of the `p:pipeinfo` element, see Section 14.7, "Processor annotations".

17. Errors

Errors in a pipeline can be divided into two classes: static errors and dynamic errors.

17.1. Static Errors

[Definition: A *static error* is one which can be detected before pipeline evaluation is even attempted.] Examples of static errors include cycles in the pipeline graph and incorrect specification of inputs and outputs.

Static errors are fatal and must be detected before any steps are evaluated.

For a complete list of static errors, see Section F.1, "Static Errors".

17.2. Dynamic Errors

[Definition: A *dynamic error* is one which occurs while a pipeline is being evaluated (and cannot be detected before evaluation begins).] Examples of dynamic errors include references to URIs that cannot be resolved, steps which fail, and pipelines that exhaust the capacity of an implementation (such as memory or disk space).

Implementations are required to evaluate the pipeline graph according to the rules of this specification, but they may choose to optimize pipeline execution in different ways. This may cause steps to be evaluated in different orders which consequently has an impact on error detection. The detection of dynamic errors is somewhat *implementation-dependent* because the order of step execution may vary. In cases where an implementation is able to run a pipeline without evaluating a particular expression, or running a particular step, the implementation is never required evaluate the expression or run the step solely in order to determine whether doing so

causes a dynamic error. For example, if a variable is declared but never referenced, an implementation may choose whether or not to evaluate the expression which initializes the variable, which means that if evaluating the variable's initializer causes a dynamic error, some implementations will signal this error and others will not.

There are some cases where this specification requires that steps must not be executed: for example, the content of a `p:when` **must not** be executed if the `test` condition is false. This means that an implementation **must not** signal any dynamic errors that would arise if the contents of the `p:when` were executed.

An implementation may signal a dynamic error before any source document is available, but only if it can determine that the error would be signaled for every possible source document and every possible set of parameter values.

If a step fails due to a dynamic error, failure propagates upwards until either a `p:try` is encountered or the entire pipeline fails. In other words, outside of a `p:try`, step failure causes the entire pipeline to fail.

For a complete list of dynamic errors, see Section F.2, "Dynamic Errors".

17.3. Step Errors

Several of the steps in the standard and option step library can generate dynamic errors.

For a complete list of the dynamic errors raised by builtin pipeline steps, see Section F.3, "Step Errors".

A. Conformance

Conformant processors **must** implement all of the features described in this specification except those that are explicitly identified as optional.

Some aspects of processor behavior are not completely specified; those features are either *implementation-dependent* or *implementation-defined*.

[Definition: An *implementation-dependent* feature is one where the implementation has discretion in how it is performed. Implementations are not required to document or explain how *implementation-dependent* features are performed.]

[Definition: An *implementation-defined* feature is one where the implementation has discretion in how it is performed. Conformant implementations **must** document how *implementation-defined* features are performed.]

A.1. Implementation-defined features

The following features are implementation-defined:

1. It is implementation-defined what additional step types, if any, are provided. See Section 2.1, “Steps”.
2. The level of support for typed values in XDM instances in an XProc pipeline is implementation-defined. See Section 3.2.1, “XML Documents”.
3. It is implementation-defined whether other media types not mentioned in this document are treated as text media types as well. See Section 3.2.3, “Text Documents”.
4. Serialization of other kinds of documents is implementation-defined. See Section 3.2.5, “Other documents”.
5. It is implementation-defined if a processor accepts any other content type shortcuts. See Section 3.4, “Specifying content types”.
6. How inputs are connected to documents outside the pipeline is implementation-defined. See Section 4, “Inputs and Outputs”.
7. How pipeline outputs are connected to documents outside the pipeline is implementation-defined. See Section 4, “Inputs and Outputs”.
8. In Version 3.0 of XProc, how (or if) implementers provide local resolution mechanisms and how (or if) they provide access to intermediate results by URI is implementation-defined. See Section 4.1, “External Documents”.
9. Except for cases which are specifically called out in , the extent to which namespace fixup, and other checks for outputs which cannot be serialized, are performed on intermediate outputs is implementation-defined. See Section 6.2, “Namespace Fixup on XML Outputs”.

10. There may be an implementation-defined mechanism for providing default values for static p:options. If such a mechanism exists, the values provided must match the sequence type declared for the option, if such a declaration exists. See Section 7, “Initiating a pipeline”.
11. The exact format of the language string is implementation-defined but should be consistent with the xml:lang attribute. See Section 8.1, “System Properties”.
12. It is implementation-defined which additional system properties are available during static analysis. See Section 8.1, “System Properties”.
13. It is implementation-defined if the processor supports any other XPath extension functions. See Section 8.11, “Other XPath Extension Functions”.
14. The value of the any other XPath extension functions during static analysis is implementation-defined. See Section 8.11, “Other XPath Extension Functions”.
15. Whether or not the pipeline processor supports passing PSVI annotations between steps is implementation-defined. See Section 9, “PSVIs in XProc”.
16. The exact PSVI properties that are preserved when documents are passed between steps is implementation-defined. See Section 9, “PSVIs in XProc”.
17. It is implementation-defined what PSVI properties, if any, are produced by extension steps. See Section 9, “PSVIs in XProc”.
18. Whether or not an extension attribute permits attribute value templates is implementation-defined. See Section 10.1, “Attribute Value Templates”.
19. How outside values are specified for pipeline options on the pipeline initially invoked by the processor is implementation-defined. See Section 11.2, “Options”.
20. The extent to which an implementation validates the lexical form of the xs:anyURI is implementation-defined. See Section 11.5.2, “Special rules for casting URIs”.
21. Support for pipeline documents written in XML 1.1 and pipeline inputs and outputs that use XML 1.1 is implementation-defined. See Section 14, “Syntax Overview”.
22. It is implementation-defined if any processing instructions are significant to an implementation. See Section 14, “Syntax Overview”.
23. The semantics of p:pipeinfo elements are implementation-defined. See Section 14.7, “Processor annotations”.

24. It is implementation-defined whether a processor supports timeouts, and if it does, how precisely and precisely how the execution time of a step is measured. See Section 14.9.4, “Controlling long running steps”.

25. Precisely what “made available” means is implementation-defined. See Section 14.9.5, “Status and debugging output”.

26. The set of URI schemes actually supported is implementation-defined. See Section 14.11, “Common errors”.

27. The presence of other compound steps is implementation-defined; XProc provides no standard mechanism for defining them or describing what they can contain. See Section 15.8.2, “Extension Steps”.

28. The default value of any serialization parameters not specified on a particular output is implementation-defined. See Section 16.3.1, “Serialization parameters”.

29. Implementations may support other method values but their results are implementation-defined. See Section 16.3.1.1, “Serialization method”.

30. The serialization method for documents with other media types is implementation-defined. See Section 16.3.1.1, “Serialization method”.

31. The precise details about what XPath expressions are allowed (for example, can the expression declare a function) is implementation-defined. See Section 16.4.1, “p:variable”.

32. The precise details about what XPath expressions are allowed (for example, can the expression declare a function) is implementation-defined. See Section 16.4.2, “p:option”.

33. When a declared step is evaluated directly by the XProc processor (as opposed to occurring as an atomic step in some container), how the input and output ports are connected to documents is implementation-defined. See Section 16.5, “p:declare-step”.

34. If no type is specified, the way that the processor determines the type of the library is implementation-defined. See Section 16.8, “p:import-functions”.

35. Whether or not a processor can import functions, and if it can, what kinds of function libraries it can import from is implementation-defined. See Section 16.8, “p:import-functions”.

36. An implementation may support encodings other than base64, but these encodings and their names are implementation-defined. See Section 16.10, “p:inline”.
37. How a processor interprets other media types is implementation-defined. See Section 16.10.4, “Other inline content”.
38. It is implementation-defined whether additional information items and properties, particularly those made available in the PSVI, are preserved between steps. See Section A.3, “Infoset Conformance”.
39. The version of Unicode supported is implementation-defined, but it is recommended that the most recent version of Unicode be used. See Section B.1, “Processor XPath Context”.
40. The context item used for binary documents is implementation-defined. See Section B.1, “Processor XPath Context”.
41. The point in time returned as the current dateTime is implementation-defined. See Section B.1, “Processor XPath Context”.
42. The implicit timezone is implementation-defined. See Section B.1, “Processor XPath Context”.
43. The default language is implementation-defined. See Section B.1, “Processor XPath Context”.
44. The default calendar is implementation-defined. See Section B.1, “Processor XPath Context”.
45. The default place is implementation-defined. See Section B.1, “Processor XPath Context”.
46. The list of available environment variables is implementation-defined. See Section B.1, “Processor XPath Context”.
47. The implicit timezone is implementation-defined. See Section B.2, “Step XPath Context”.
48. The default language is implementation-defined. See Section B.2, “Step XPath Context”.
49. The default calendar is implementation-defined. See Section B.2, “Step XPath Context”.

50. The default place is implementation-defined. See Section B.2, “Step XPath Context”.
51. The list of available environment variables is implementation-defined. See Section B.2, “Step XPath Context”.

A.2. Implementation-dependent features

The following features are implementation-dependent:

1. The evaluation order of steps not connected to one another is implementation-dependent. See Section 2, “Pipeline Concepts”.
2. The underlying representations of other kinds of documents are implementation-dependent. See Section 3.2.5, “Other documents”.
3. Outside of a try/catch, the disposition of error messages is implementation-dependent See Section 4, “Inputs and Outputs”.
4. Resolving a URI locally may involve resolvers of various sorts and possibly appeal to implementation-dependent mechanisms such as catalog files. See Section 4.1, “External Documents”.
5. Whether (and when and how) or not the intermediate results that pass between steps are ever written to a filesystem is implementation-dependent. See Section 4.1, “External Documents”.
6. Which steps are forbidden, what privileges are needed to access resources, and under what circumstances these security constraints apply is implementation-dependent. See Section 12, “Security Considerations”.
7. The error codes that appear in cause are implementation-dependent. See Section 15.7.3.2, “c:error”.
8. It is implementation-dependent whether or not atomic steps can be defined through some other means. See Section 15.8, “Atomic Steps”.
9. Whether or not an implementation allows users to provide their own external steps is implementation-dependent. See Section 16.5, “p:declare-step”.
10. Implementations may use extension attributes to provide implementation-dependent information about a declared step. See Section 16.5.2, “Declaring external steps”.

11. The detection of dynamic errors is somewhat implementation-dependent because the order of step execution may vary. See Section 17.2, "Dynamic Errors".
12. The set of available documents (those that may be retrieved with a URI) is implementation-dependent. See Section B.1, "Processor XPath Context".
13. The set of available text resources (those that may be retrieved with a URI) is implementation-dependent. See Section B.1, "Processor XPath Context".
14. The set of available collections is implementation-dependent. See Section B.1, "Processor XPath Context".
15. The set of available URI collections is implementation-dependent. See Section B.1, "Processor XPath Context".
16. The default URI collection is implementation-dependent. See Section B.1, "Processor XPath Context".
17. The set of available documents (those that may be retrieved with a URI) is implementation-dependent. See Section B.2, "Step XPath Context".
18. The set of available text resources (those that may be retrieved with a URI) is implementation-dependent. See Section B.2, "Step XPath Context".
19. The set of available URI collections is implementation-dependent. See Section B.2, "Step XPath Context".
20. The default URI collection is implementation-dependent. See Section B.2, "Step XPath Context".

A.3. Infoset Conformance

This specification conforms to the XML Information Set [Infoset]. The information corresponding to the following information items and properties must be available to the processor for the documents that flow through the pipeline.

- The `Document Information Item` with `[base URI]` and `[children]` properties.

- `Element Information Items` with `[base URI]`, `[children]`, `[attributes]`, `[in-scope namespaces]`, `[prefix]`, `[local name]`, `[namespace name]`, `[parent]` properties.
- `Attribute Information Items` with `[namespace name]`, `[prefix]`, `[local name]`, `[normalized value]`, `[attribute type]`, and `[owner element]` properties.
- `Character Information Items` with `[character code]`, `[parent]`, and, optionally, `[element content whitespace]` properties.
- `Processing Instruction Information Items` with `[base URI]`, `[target]`, `[content]` and `[parent]` properties.
- `Comment Information Items` with `[content]` and `[parent]` properties.
- `Namespace Information Items` with `[prefix]` and `[namespace name]` properties.

It is *implementation-defined* whether additional information items and properties, particularly those made available in the PSVI, are preserved between steps.

B. XPath contexts in XProc

Two kinds of XPath context are relevant in XProc: the context of the pipeline itself (Section B.1, "Processor XPath Context") and the context *within* steps (Section B.2, "Step XPath Context").

B.1. Processor XPath Context

When the XProc processor evaluates an XPath expression using XPath, unless otherwise indicated by a particular step, it does so with the following static context:

XPath 1.0 compatibility mode
: False

Statically known namespaces
: The namespace declarations in-scope for the containing element.

Default element/type namespace
The null namespace.

Default function namespace
The default function namespace is `http://www.w3.org/2005/xpath-functions`, as defined in [XPath and XQuery Functions and Operators 3.1]. Function names that do not contain a colon always refer to the default function namespace, any in-scope binding for the default namespace *does not* apply. This specification does not provide a mechanism to override the default function namespace.

In-scope schema definitions
A basic XPath 3.1 XProc processor includes the following named type definitions in its in-scope schema definitions:

- All the primitive atomic types defined in [W3C XML Schema: Part 2], with the exception of `xs:NOTATION`. That is: `xs:anyAtomicType`, `xs:anySimpleType`, `xs:anyURI`, `xs:base64Binary`, `xs:boolean`, `xs:date`, `xs:dateTime`, `xs:decimal`, `xs:double`, `xs:duration`, `xs:float`, `xs:gDay`, `xs:gMonth`, `xs:gMonthDay`, `xs:gYear`, `xs:gYearMonth`, `xs:hexBinary`, `xs:QName`, `xs:string`, and `xs:time`.

- The derived atomic type `xs:integer` defined in [W3C XML Schema: Part 2].

- The types `xs:anyType`, `xs:yearMonthDuration`, `xs:dayTimeDuration`, `xs:untyped`, and `xs:untypedAtomic` defined in [XQuery and XPath Data Model 3.1].

In-scope variables
Variables and options are lexically scoped. The union of the options and the variables that are "visible" from the step's lexical position are available as variable bindings to the XPath processor. Variables and options can shadow each other, only the lexically most recent bindings are visible.

Context item static type
Document.

Function signatures
The signatures of the [XPath and XQuery Functions and Operators 3.1] in namespaces `http://www.w3.org/2005/xpath-functions`, `http://www.w3.org/2005/xpath-functions/math`, `http://www.w3.org/2005/xpath-functions/map` and `http://www.w3.org/2005/xpath-functions/array`. Additionally the function signatures defined in Section 8, "XPath Extension Functions".

Statically known collations
Implementation-defined but **must** include the Unicode code point collation. The version of Unicode supported is *implementation-defined*, but it is recommended that the most recent version of Unicode be used.

Default collation
Unicode code point collation.

Static base URI
The base URI of the element on which the expression occurs.

Statically known documents
None.

Statically known collections
None.

Statically known default collection type
`item()*`

Statically known decimal formats
None.

And the following dynamic context:

context item
The context item. The context item is either specified with a *connection* or is taken from the *default readable port*. If the explicit connection or the default readable port provides no or more than one document then the context item is undefined. It is a *dynamic error* (`err:XD0001`) if the XPath expression makes use of the context item, but the context item is undefined.

The context item used for an XML, text, or JSON document is the XDM representation of that item. The context item used for binary documents is *implementation-defined*.

If there is no explicit connection and there is no default readable port then the context item is undefined.

context position and context size
If the context item is defined, the context position and context size are both "1". It is a *dynamic error* (`err:XD0001`) to refer to the context position or size if the context item is undefined.

Variable values
The union of the in-scope options and variables are available as variable bindings to the XPath processor.

Named functions
The [XPath and XQuery Functions and Operators 3.1] and the Section 8, "XPath Extension Functions".

Current dateTime
The point in time returned as the current dateTime is *implementation-defined*.

Implicit timezone
The implicit timezone is *implementation-defined*.

Default language
The default language is *implementation-defined*.

Default calendar
The default calendar is *implementation-defined*.

Default place
The default place is *implementation-defined*.

Available documents
The set of available documents (those that may be retrieved with a URI) is *implementation-dependent*.

Available text resources
The set of available text resources (those that may be retrieved with a URI) is *implementation-dependent*.

Available collections
The set of available collections is *implementation-dependent*.

Default collection
None.

Available URI collections
The set of available URI collections is *implementation-dependent*.

Default URI collection
The default URI collection is *implementation-dependent*.

Environment variables
The list of available environment variables is *implementation-defined*.

B.2. Step XPath Context

When a step evaluates an XPath expression using XPath 3.1, unless otherwise indicated by a particular step, it does so with the following static context:

XPath 1.0 compatibility mode
False

Statically known namespaces
The namespace declarations in-scope for the containing element.

Default element/type namespace
The null namespace.

Default function namespace
The default function namespace is `http://www.w3.org/2005/xpath-functions`, as defined in [XPath and XQuery Functions and Operators 3.1]. Function names that do not contain a colon always refer to the default function namespace, any in-scope binding for the default namespace *does not* apply. This specification does not provide a mechanism to override the default function namespace.

In-scope schema definitions
The same as the Section B.1, “Processor XPath Context”.

In-scope variables
None, unless otherwise specified by the step.

Context item static type
Document.

Function signatures
The signatures of the [XPath and XQuery Functions and Operators 3.1] in namespaces `http://www.w3.org/2005/xpath-functions`, `http://www.w3.org/2005/xpath-functions/math`, `http://www.w3.org/2005/xpath-functions/map` and `http://www.w3.org/2005/xpath-functions/array`.

Statically known collations
Implementation-defined but **must** include the Unicode code point collation.

Default collation
Unicode code point collation.

Static base URI
The base URI of the element on which the expression occurs.

Statically known documents
None.

Statically known collections
None.

Statically known default collection type
item()*

Statically known decimal formats
None.

And the following initial dynamic context:

context item
The document node of the document that appears on the primary input of the step, unless otherwise specified by the step.

context position and context size
The context position and context size are both "1", unless otherwise specified by the step.

Variable values
None, unless otherwise specified by the step.

Named functions
The [XPath and XQuery Functions and Operators 3.1].

Current dateTime
An implementation-defined point in time.

Implicit timezone
The implicit timezone is *implementation-defined*.

Default language
The default language is *implementation-defined*.

Default calendar
The default calendar is *implementation-defined*.

Default place
The default place is *implementation-defined*.

Available documents
The set of available documents (those that may be retrieved with a URI) is *implementation-dependent*.

Available text resources
The set of available text resources (those that may be retrieved with a URI) is *implementation-dependent*.

Available collections
None.

Default collection
None.

Available URI collections
The set of available URI collections is *implementation-dependent*.

Default URI collection
The default URI collection is *implementation-dependent*.

Environment variables
The list of available environment variables is *implementation-defined*.

> **Note**
>
> Some steps may also provide for implementation-defined or implementation-dependent amendments to the contexts. Those amendments are in addition to any specified by XProc.

C. References

C.1. Normative References

[Steps 3.0] *XProc 3.0: Standard Step Library*. Norman Walsh, Achim Berndzen, Gerrit Imsieke and Erik Siegel, editors.

[Infoset] *XML Information Set (Second Edition)*. John Cowan, Richard Tobin, editors. W3C Working Group Note 04 February 2004.

[XML 1.0] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, et. al. editors. W3C Recommendation 26 November 2008.

[Namespaces 1.0] *Namespaces in XML 1.0 (Third Edition)*. Tim Bray, Dave Hollander, Andrew Layman, et. al., editors. W3C Recommendation 8 December 2009.

[XML 1.1] *Extensible Markup Language (XML) 1.1 (Second Edition)*. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, et. al. editors. W3C Recommendation 16 August 2006.

[Namespaces 1.1] *Namespaces in XML 1.1 (Second Edition)*. Tim Bray, Dave Hollander, Andrew Layman, et. al., editors. W3C Recommendation 16 August 2006.

[XPath 3.1] *XML Path Language (XPath) 3.1*. Jonathan Robie, Michael Dyck, Josh Spiegel, editors. W3C Recommendation. 21 March 2017.

[XQuery and XPath Data Model 3.1] *XQuery and XPath Data Model 3.1*. Norman Walsh, John Snelson, and Andrew Coleman, editors. W3C Recommendation. 21 March 2017.

[Serialization] *XSLT and XQuery Serialization 3.1*. Andrew Coleman and C. M. Sperberg-McQueen, editors. W3C Recommendation. 21 March 2017.

[XPath and XQuery Functions and Operators 3.1] *XPath and XQuery Functions and Operators 3.1*. Michael Kay, editor. W3C Recommendation. 21 March 2017

[XSLT 3.0] *XSL Transformations (XSLT) Version 3.0*. Michael Kay, editor. W3C Recommendation. 8 June 2017.

[XQuery 1.0] *XQuery 1.0: An XML Query Language*. Scott Boag, Don Chamberlin, Mary Fernández, et. al., editors. W3C Recommendation. 23 January 2007.

[W3C XML Schema: Part 1] *XML Schema Part 1: Structures Second Edition*. Henry S. Thompson, David Beech, Murray Maloney, et. al., editors. World Wide Web Consortium, 28 October 2004.

[W3C XML Schema: Part 2] *XML Schema Part 2: Datatypes Second Edition*. Paul V. Biron and Ashok Malhotra, editors. World Wide Web Consortium, 28 October 2004.

[xml:id] *xml:id Version 1.0*. Jonathan Marsh, Daniel Veillard, and Norman Walsh, editors. W3C Recommendation. 9 September 2005.

[XML Base] *XML Base (Second Edition)*. Jonathan Marsh and Richard Tobin, editors. W3C Recommendation. 28 January 2009.

[RFC 2119] *Key words for use in RFCs to Indicate Requirement Levels*. S. Bradner. Network Working Group, IETF, Mar 1997.

[RFC 2396] *Uniform Resource Identifiers (URI): Generic Syntax*. T. Berners-Lee, R. Fielding, and L. Masinter. Network Working Group, IETF, Aug 1998.

[RFC 3023] *RFC 3023: XML Media Types*. M. Murata, S. St. Laurent, and D. Kohn, editors. Internet Engineering Task Force. January, 2001.

[RFC 2046] *RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. N. Freed, N. Borenstein, editors. Internet Engineering Task Force. November, 1996.

[RFC 3986] *RFC 3986: Uniform Resource Identifier (URI): General Syntax*. T. Berners-Lee, R. Fielding, and L. Masinter, editors. Internet Engineering Task Force. January, 2005.

D. Glossary

HTML media type

The "`text/html`" and "`application/xhtml+xml`" media types are *HTML media types*.

JSON media type

The "`application/json`" media type and all media types of the form "`application/`*`something`*`+json`" are *JSON media types*.

Namespaces in XML

Unless otherwise noted, the term *Namespaces in XML* refers equally to [Namespaces 1.0] and [Namespaces 1.1].

XML

XProc is intended to work equally well with [XML 1.0] and [XML 1.1]. Unless otherwise noted, the term "*XML*" refers equally to both versions.

XML media type

The "`application/xml`" and "`text/xml`" media types and all media types of the form "*`something/something`*`+xml`" (except for "`application/xhtml+xml`" which is explicitly an *HTML media type*) are *XML media types*.

ancestors

The *ancestors* of a step, if it has any, are its *container* and the ancestors of its container.

anonymous input

The *compound steps* `p:for-each` and `p:viewport` each declare a single primary input without a port name. Such an input is called an *anonymous input*.

atomic step

An *atomic step* is a step that does not contain a subpipline when it is invoked.

attribute value template

In an attribute that is designated as an *attribute value template*, an expression can be used by surrounding the expression with curly brackets (`{}`), following the general rules for *value templates*

bag-merger

The *bag-merger* of two or more bags (where a bag is an unordered list or, equivalently, something like a set except that it may contain duplicates) is a bag

constructed by starting with an empty bag and adding each member of each of the input bags in turn to it. It follows that the cardinality of the result is the sum of the cardinality of all the input bags.

by source
A document is specified *by source* if it references a specific port on another step.

by URI
A document is specified *by URI* if it is referenced with a URI.

compound step
A *compound step* is a step that contains one or more *subpipelines*.

connection
A *connection* associates an input or output port with some data source.

contained steps
The steps that occur directly within a container are called that step's *contained steps*. In other words, "container" and "contained steps" are inverse relationships.

container
A *container* is either a compound step or one of the non-step wrapper elements in a compound step that contains several subpipelines.

declared inputs
The input ports declared on a step are its *declared inputs*.

declared outputs
The output ports declared on a step are its *declared outputs*.

default readable port
The *default readable port*, which may be undefined, is a specific step name/port name pair from the set of readable ports.

document
A *document* is a *representation* and its *document properties*.

document properties
The *document properties* are key/value pairs; they are exposed to the XProc pipeline as a map (`map(xs:QName, item()*)`).

dynamic error
: A *dynamic error* is one which occurs while a pipeline is being evaluated (and cannot be detected before evaluation begins).

dynamic evaluation
: *Dynamic evaluation* consists of tasks which, in general, cannot be performed out until a source document is available.

effectively excluded
: If the effective boolean value of the `[p:]use-when` expression is false, then the element and all of its descendants are *effectively excluded* from the pipeline document.

empty environment
: The *empty environment* contains no readable ports, an undefined default readable port, and no in-scope bindings.

empty sequence
: An *empty sequence* of documents is specified with the `p:empty` element.

environment
: The *environment* is a context-dependent collection of information available within subpipelines.

extension attribute
: An element from the XProc namespace **may** have any attribute not from the XProc namespace, provided that the expanded-QName of the attribute has a non-null namespace URI. Such an attribute is called an *extension attribute*.

external step
: An *external step* is one supported by the implementation, but which has no exposed subpipeline.

implementation-defined
: An *implementation-defined* feature is one where the implementation has discretion in how it is performed. Conformant implementations **must** document how *implementation-defined* features are performed.

implementation-dependent
: An *implementation-dependent* feature is one where the implementation has discretion in how it is performed. Implementations are not required to document or explain how *implementation-dependent* features are performed.

in-scope bindings
: The *in-scope bindings* are a set of name-value pairs, based on *option* and *variable* bindings.

inherited environment
: The *inherited environment* of a *contained step* is an environment that is the same as the environment of its *container* with the standard modifications.

initial environment
: An *initial environment* is a *connection* for each of the *readable ports* and a set of option bindings used to construct the initial *in-scope bindings*.

inline document
: An *inline document* is specified directly in the body of the element to which it connects.

last step
: The *last step* in a subpipeline is its last step in document order.

map attribute
: A *map attribute* is an option's syntactic shortcut attribute for which the option's sequence type is a map or array.

matches
: A step *matches* its signature if and only if it specifies an input for each declared input, it specifies no inputs that are not declared, it specifies an option for each option that is declared to be required, and it specifies no options that are not declared.

namespace fixup
: To produce a serializable *XML* document, the XProc processor must sometimes add additional namespace nodes, perhaps even renaming prefixes, to satisfy the constraints of *Namespaces in XML*. This process is referred to as *namespace fixup*.

option
: An *option* is a name/value pair. The name **must** be an expanded name. The value may be any XPath data model value.

pipeline
: A *pipeline* is a set of connected steps, with outputs of one step flowing into inputs of another.

primary input port
: If a step has an input port which is explicitly marked "`primary='true'`", or if it has exactly one document input port and that port is *not* explicitly marked "`primary='false'`", then that input port is the *primary input port* of the step.

primary output port
: If a step has an output port which is explicitly marked "`primary='true'`", or if it has exactly one document output port and that port is *not* explicitly marked "`primary='false'`", then that output port is the *primary output port* of the step.

prologue
: The *prologue* consists of the `p:input`, `p:output`, and `p:option` elements.

readable ports
: The *readable ports* are a set of step name/port name pairs.

representation
: A *representation* is a data structure used by an XProc processor to refer to the actual document content.

selection pattern
: A *selection pattern* uses a subset of the syntax for path expressions, and is defined to match a node if the corresponding path expression would select the node. It is defined as in the XSLT 3.0 specification.

shadow
: We say that a variable *shadows* another variable (or option) if it has the same name and appears later in the same lexical scope.

signature
: The *signature* of a step is the set of inputs, outputs, and options that it is declared to accept.

static analysis
: *Static analysis* consists of those tasks that can be performed by inspection of the pipeline alone, including the binding of static options, computation of serialization properties and document-properties, evaluation of `use-when` expressions, performing a static analysis of all XPath expressions, and detecting static errors.

static error

A *static error* is one which can be detected before pipeline evaluation is even attempted.

step

A *step* is the basic computational unit of a pipeline.

step type exports

The *step type exports* of an XProc element, against the background of a set of URIs of resources already visited (call this set *Visited*), are defined by cases.

subpipeline

Sibling steps and variables (and the connections between them) form a *subpipeline*.

text media type

Media types of the form "`text/something`" are *text media types* with the exception of "`text/xml`" which is an XML media type, and "`text/html`" which is an HTML media type. Additionally the media types "`application/javascript`", "`application/relax-ng-compact-syntax`", and "`application/xquery`" are also text media types.

text value template

In a text node that is designated as a *text value template*, expressions can be used by surrounding each expression with curly brackets (`{}`), following the general rules for *value templates*.

value template

Collectively, attribute value templates and text value templates are referred to as *value templates*.

variable

A *variable* is a name/value pair. The name **must** be an expanded name. The value may be any XPath data model value.

visible

If two names are in the same scope, we say that they are *visible* to each other.

E. Pipeline Language Summary

This appendix summarizes the XProc pipeline language. Machine readable descriptions of this language are available in RELAX NG (and the RELAX NG compact syntax).

```
<p:for-each
  name? = NCName>
    ((p:with-input? &
      p:output*),
     subpipeline)
</p:for-each>
```

```
<p:viewport
  name? = NCName
  match = XSLTSelectionPattern>
    ((p:with-input? &
      p:output?),
     subpipeline)
</p:viewport>
```

```
<p:choose
  name? = NCName>
    (p:with-input?,
     ((p:when+,
       p:otherwise?) |
      (p:when*,
       p:otherwise)))
</p:choose>
```

```
<p:when
  name? = NCName
  test = XPathExpression
  collection? = boolean>
    (p:with-input?,
     p:output*,
     subpipeline)
</p:when>
```

```
<p:otherwise
  name? = NCName>
    (p:output*,
     subpipeline)
</p:otherwise>
```

```
<p:if
  name? = NCName
  test = XPathExpression
  collection? = boolean>
    (p:with-input?,
     p:output*,
     subpipeline)
</p:if>
```

```
<p:group
  name? = NCName>
    (p:output*,
     subpipeline)
</p:group>
```

```
<p:try
  name? = NCName>
    (p:output*,
     subpipeline,
     ((p:catch+,
        p:finally?) |
       (p:catch*,
        p:finally)))
</p:try>
```

```
<p:catch
  name? = NCName
  code? = EQNameList>
    (p:output*,
     subpipeline)
</p:catch>
```

```
<p:finally
  name? = NCName>
    (p:output*,
     subpipeline)
</p:finally>
```

```
<p:atomic-step
  name? = NCName>
    (p:with-input |
     p:with-option)*
</p:atomic-step>
```

```
<ext:atomic-step
  name? = NCName>
    (p:with-input |
     p:with-option)*
</ext:atomic-step>
```

```
<p:input
  port = NCName
  sequence? = boolean
  primary? = boolean
  select? = XPathExpression
  content-types? = ContentTypes
  href? = { anyURI }
  exclude-inline-prefixes? = ExcludeInlinePrefixes>
    ((p:empty |
       (p:document |
        p:inline)*) |
     anyElement*)
</p:input>
```

```
<p:with-input
  port? = NCName
  select? = XPathExpression
  href? = { anyURI }
  pipe? = string
  exclude-inline-prefixes? = ExcludeInlinePrefixes>
    ((p:empty |
       (p:document |
        p:pipe |
        p:inline)*) |
     anyElement*)
</p:with-input>
```

```
<p:output
  port? = NCName
  sequence? = boolean
  primary? = boolean
  content-types? = ContentTypes />
```

```
<p:output
  port? = NCName
  sequence? = boolean
  primary? = boolean
  content-types? = ContentTypes
  href? = { anyURI }
  pipe? = string
  exclude-inline-prefixes? = ExcludeInlinePrefixes>
    ((p:empty |
       (p:document |
        p:pipe |
        p:inline)*) |
     anyElement*)
</p:output>
```

```
<p:output
  port? = NCName
  sequence? = boolean
  primary? = boolean
  content-types? = ContentTypes
  href? = { anyURI }
  pipe? = string
  exclude-inline-prefixes? = ExcludeInlinePrefixes
  serialization? = map(xs:QName,item()*)>
    ((p:empty |
       (p:document |
        p:pipe |
        p:inline)*) |
     anyElement*)
</p:output>
```

```
<p:variable
  name = EQName
  as? = XPathSequenceType
  select = XPathExpression
  collection? = boolean
  href? = { anyURI }
  pipe? = string
  exclude-inline-prefixes? = ExcludeInlinePrefixes>
    ((p:empty |
       (p:document |
        p:pipe |
        p:inline)*) |
     anyElement*)
</p:variable>
```

```
<p:option
  name = EQName
  as? = XPathSequenceType
  values? = string
  static? = boolean
  required? = boolean
  select? = XPathExpression
  visibility? = private|public />
```

```
<p:with-option
  name = EQName
  as? = XPathSequenceType
  select = XPathExpression
  collection? = boolean
  href? = { anyURI }
  pipe? = string
  exclude-inline-prefixes? = ExcludeInlinePrefixes>
    ((p:empty |
       (p:document |
        p:pipe |
        p:inline)*) |
     anyElement*)
</p:with-option>
```

```
<p:declare-step
  name? = NCName
  type? = EQName
  psvi-required? = boolean
  xpath-version? = decimal
  exclude-inline-prefixes? = ExcludeInlinePrefixes
  version? = 3.0
  visibility? = private|public>
    (p:import |
     p:import-functions)*,
    (p:input |
     p:output |
     p:option)*,
    p:declare-step*,
    subpipeline?
</p:declare-step>
```

```
<p:declare-step
  name? = NCName
  type? = EQName
  psvi-required? = boolean
  xpath-version? = decimal
  exclude-inline-prefixes? = ExcludeInlinePrefixes
  version? = 3.0
  visibility? = private|public>
    (p:input |
     p:output |
     p:option)*
</p:declare-step>
```

```
<p:library
  psvi-required? = boolean
  xpath-version? = decimal
  exclude-inline-prefixes? = ExcludeInlinePrefixes
  version? = 3.0>
    (p:import |
     p:import-functions)*,
    p:option*,
    p:declare-step*
</p:library>
```

```
<p:import
  href = anyURI />
```

```
<p:import-functions
  href = anyURI
  content-type? = ContentType
  namespace? = string />
```

```
<p:pipe
  step? = NCName
  port? = NCName />
```

```
<p:inline
  exclude-inline-prefixes? = ExcludeInlinePrefixes
  content-type? = string
  document-properties? = map(xs:QName,item()*)
  encoding? = string>
    anyNode*
</p:inline>
```

```
<p:document
  href = { anyURI }
  content-type? = string
  document-properties? = map(xs:QName,item()*)
  parameters? = map(xs:QName,item()*) />
```

```
<p:empty />
```

```
<p:documentation>
    any-well-formed-content*
</p:documentation>
```

```
<p:pipeinfo>
    any-well-formed-content*
</p:pipeinfo>
```

The core steps are also summarized here.

As are the optional steps.

And the step vocabulary elements.

F. List of Error Codes

The following error codes are defined by this specification.

F.1. Static Errors

The following *static errors* are defined:

Static Errors

`err:XS0001`
: It is a static error if there are any loops in the connections between steps, variables, and options: no step, variable, or option can be connected to itself nor can there be any sequence of connections through other steps that leads back to itself.

  See: Connections

`err:XS0002`
: All steps in the same scope must have unique names: it is a static error if two steps with the same name appear in the same scope.

  See: Scoping of step names

`err:XS0003`
: It is a static error if any declared input is not connected.

  See: Inputs and Outputs

`err:XS0004`
: It is a static error to declare two or more options on the same step with the same name.

  See: p:option

`err:XS0006`
: It is a static error if the primary output port has no explicit connection and the last step in the subpipeline does not have a primary output port.

  See: p:for-each, p:viewport, Declaring pipelines

`err:XS0008`
: It is a static error if any element in the XProc namespace has attributes not defined by this specification unless they are extension attributes.

  See: Common errors

`err:XS0010`
: It is a static error if a pipeline contains a step whose specified inputs, outputs, and options do not match the signature for steps of that type.

See: Extension Steps

`err:XS0011`
It is a static error to identify two ports with the same name on the same step.

See: p:input, p:output

`err:XS0014`
It is a static error to identify more than one output port as primary.

See: p:output

`err:XS0015`
It is a static error if a compound step has no contained steps.

See: Common errors

`err:XS0017`
It is a static error to specify that an option is both required and has a default value.

See: p:option

`err:XS0018`
If an option is required, it is a static error to invoke the step without specifying a value for that option.

See: p:option

`err:XS0022`
In all cases except when the p:pipe is within an p:output of a compound step, it is a static error if the port identified by the p:pipe is not in the readable ports of the step that contains the p:pipe.

See: p:pipe

`err:XS0025`
It is a static error if the expanded-QName value of the type attribute is in no namespace or in the XProc namespace.

See: Declaring pipelines

`err:XS0027`
: It is a static error if an option is specified with both the shortcut form and the long form.

  See: Syntactic Shortcut for Option Values

`err:XS0028`
: It is a static error to declare an option or variable in the XProc namespace.

  See: p:variable, p:option

`err:XS0029`
: It is a static error to specify a connection for a p:output inside a p:declare-step for an external step.

  See: p:output

`err:XS0030`
: It is a static error to specify that more than one input port is the primary.

  See: p:input

`err:XS0031`
: It is a static error to use an option name in p:with-option if the step type being invoked has not declared an option with that name.

  See: p:with-option, Syntactic Shortcut for Option Values

`err:XS0032`
: It is a static error if no connection is provided and the default readable port is undefined.

  See: p:with-input, Connection precedence, Connection precedence

`err:XS0036`
: All the step types in a pipeline or library must have unique names: it is a static error if any step type name is built-in and/or declared or defined more than once in the same scope.

  See: Scoping of step type names, Handling Circular and Re-entrant Library Imports (Non-Normative), Handling Circular and Re-entrant Library Imports (Non-Normative), Handling Circular and Re-entrant Library Imports (Non-Normative)

`err:XS0037`
: It is a static error if any user extension step or any element in the XProc namespace other than p:inline directly contains text nodes that do not consist entirely of whitespace.

  See: Common errors

`err:XS0038`
: It is a static error if any required attribute is not provided.

  See: Common errors

`err:XS0043`
: It is a static error to specify a port name on p:with-input for p:for-each, p:viewport, p:choose, p:when, or p:if.

  See: p:with-input

`err:XS0044`
: It is a static error if any step contains an atomic step for which there is no visible declaration.

  See: Common errors

`err:XS0048`
: It is a static error to use a declared step as a compound step.

  See: Extension Steps

`err:XS0052`
: It is a static error if the URI of a p:import cannot be retrieved or if, once retrieved, it does not point to a p:library or p:declare-step.

  See: p:import

`err:XS0053`
: It is a static error to import a single pipeline if that pipeline does not have a type.

  See: p:import

`err:XS0057`
: It is a static error if the exclude-inline-prefixes attribute does not contain a list of tokens or if any of those tokens (except #all or #default) is not a prefix bound to a namespace in the in-scope namespaces of the element on which it occurs.

  See: Inline XML and HTML content

`err:XS0058`
: It is a static error if the value #default is used within the exclude-inline-prefixes attribute and there is no default namespace in scope.

  See: Inline XML and HTML content

`err:XS0059`
: It is a static error if the pipeline element is not p:declare-step or p:library.

  See: Common errors

`err:XS0060`
: It is a static error if the processor encounters an explicit request for a version of the language other than “3.0”.

  See: Versioning Considerations

`err:XS0062`
: It is a static error if a required version attribute is not present.

  See: Versioning Considerations, Declaring pipelines

`err:XS0063`
: It is a static error if the value of the version attribute is not a xs:decimal.

  See: Versioning Considerations

`err:XS0064`
: It is a static error if the code attribute is missing from any but the last p:catch or if any error code occurs in more than one code attribute among sibling p:catch elements.

  See: p:catch

`err:XS0065`
: It is a static error if there is no primary input port.

See: p:with-input

`err:XS0066`
It is a static error if an expression does not have a closing right curly bracket or if an unescaped right curly bracket occurs outside of an expression.

See: Value Templates

`err:XS0067`
It is a static error if the step attribute is not specified, and there is no default readable port. It is a static error if the port attribute is not specified, and the step identified has no primary output port.

See: p:pipe

`err:XS0068`
It is a static error if the port attribute is not specified, and the step identified has no primary output port.

See: p:pipe

`err:XS0069`
It is a static error if the encoding specified is not supported by the implementation.

See: p:inline

`err:XS0071`
All the static options in a pipeline or library must have unique names: it is a static error if any static option name is declared more than once in the same scope.

See: Scoping of static option names

`err:XS0072`
It is a static error if the name of any output port on the p:finally is the same as the name of any other output port in the p:try or any of its sibling p:catch elements.

See: p:finally

`err:XS0073`
It is a static error if any specified name is not the name of an in-scope step.

See: Additional dependent connections

`err:XS0074`
It is a static error if a p:choose has neither a p:when nor a p:otherwise.

See: p:choose

`err:XS0075`
It is a static error if a p:try does not have at least one subpipeline step, at least one of p:catch or p:finally, and at most one p:finally.

See: p:try

`err:XS0076`
It is a static error if there are any loops in the connections between steps and variables: no step can refer to a variable if there is any sequence of connections from that step that leads back to the input that provides the context node for the expression that defines the value of the variable.

See: p:variable, p:with-option

`err:XS0077`
It is a static error if the value on an attribute of an XProc element does not satisfy the type required for that attribute.

See: Common errors

`err:XS0078`
When the p:pipe is within an p:output of a compound step, it is a static error if the port identified by the p:pipe is not in the readable ports of the compound step and is not a readable port of a contained step.

See: p:pipe

`err:XS0079`
It is a static error if comments, non-whitespace text nodes, or processing instructions occur as siblings of an element node that would be treated as an implicit inline.

See: Implicit inlines

`err:XS0080`
: It is a static error to include more than one p:with-option with the same option name as part of the same step invocation.

  See: p:with-option

`err:XS0081`
: If href is specified, it is a static error if any child elements other than p:documentation and p:pipeinfo are present.

  See: p:with-input

`err:XS0082`
: If pipe is specified, it is a static error any child elements other than p:documentation and p:pipeinfo are present.

  See: p:with-input

`err:XS0083`
: It is a static error if the value of the code attribute is not a whitespace separated list of EQNames.

  See: p:catch

`err:XS0085`
: It is a static error if both a href attribute and a pipe attribute are present.

  See: p:with-input, p:with-input

`err:XS0086`
: It is a static error to provide more than one p:with-input for the same port.

  See: p:with-input

`err:XS0087`
: It is a static error if the name attribute on p:option or p:variable has a prefix which is not bound to a namespace.

  See: p:variable, p:option

`err:XS0088`
: It is a static error if the qualified name of a p:variable shadows the name of a static option.

See: p:variable, p:option

`err:XS0089`
It is a static error if the p:empty binding appears as a sibling of any other binding, including itself.

See: p:empty

`err:XS0090`
It is a static error if the value of the pipe attribute contains any tokens not of the form port-name, port-name@step-name, or @step-name.

See: p:with-input

`err:XS0091`
It is a static error if an p:option shadows another option declared within the same p:declare-step.

See: Declaring pipelines

`err:XS0092`
It is a static error if a p:with-option attempts to change the value of an option that is declared static.

See: p:with-option, Syntactic Shortcut for Option Values

`err:XS0094`
It is a static error if a p:variable does not have a select attribute.

See: p:variable

`err:XS0095`
It is a static error to specify that an option is both required and static.

See: p:option

`err:XS0096`
It is a static error if the sequence type is not syntactically valid.

See: Variable and option types

`err:XS0097`
: It is a static error if an attribute in the XProc namespace appears on an element in the XProc namespace.

  See: Common Attributes

`err:XS0099`
: It is a static error if step or port are not valid instances of NCName.

  See: p:pipe

`err:XS0100`
: It is a static error if the pipeline document does not conform to the grammar for pipeline documents.

  See: Common errors

`err:XS0101`
: It is a static error if the values list is not an XPath sequence of atomic values.

  See: p:option

`err:XS0102`
: It is a static error if alternative subpipelines have different primary output ports.

  See: p:choose, p:try

`err:XS0103`
: It is a static error if the URI of a p:import-functions element cannot be retrieved or if, once retrieved, it points to a library that the processor cannot import.

  See: p:import-functions

`err:XS0104`
: It is a static error if the processor cannot load the function library.

  See: p:import-functions

`err:XS0105`
: It is a static error if a function imported from a library has the same name and arity as a function already imported.

  See: p:import-functions

`err:XS0106`
: It is a static error if the processor detects that a particular library is unloadable.

  See: p:import-functions

`err:XS0107`
: It is a static error in XProc if any XPath expression or the XSLT selection pattern in option match on p:viewport contains a static error (error in expression syntax, references to unknown variables or functions, etc.).

  See: Initiating a pipeline

`err:XS0108`
: It is a static error if the p:if step does not specify a primary output port.

  See: p:if

`err:XS0109`
: It is a static error if options that are the direct children of p:library are not declared “static”

  See: Static Options

`err:XS0110`
: It is a static error if the requested XPath version is less than “3.1”

  See: Declaring pipelines, p:library

`err:XS0111`
: It is a static error if an unrecognized content type shortcut is specified.

  See: Specifying content types

`err:XS0112`
: It is a static error if p:finally declares a primary output port either explicitly or implicitly.

  See: p:finally

`err:XS0113`
: It is a static error if either [p:]expand-text or [p:]inline-expand-text is to be interpreted by the processor and it does not have the value “true” or “false”.

See: Expand text attributes

`err:XS0114`
: It is a static error if a port name is specified and the step type being invoked does not have an input port declared with that name.

  See: p:with-input

`err:XS0115`
: It is a static error if two or more elements are contained within a deadlocked network of [p:]use-when expressions.

  See: Conditional Element Exclusion

F.2. Dynamic Errors

The following *dynamic errors* are defined:

Dynamic Errors

`err:XD0001`
: It is a dynamic error if an XPath expression makes reference to the context item, size, or position when the context item is undefined.

  See: p:choose, p:when, p:if, p:variable, p:option, p:with-option, Processor XPath Context, Processor XPath Context

`err:XD0006`
: If sequence is not specified, or has the value false, then it is a dynamic error unless exactly one document appears on the declared port.

  See: p:input

`err:XD0007`
: If sequence is not specified on p:output, or has the value false, then it is a dynamic error if the step does not produce exactly one document on the declared port.

  See: p:output

`err:XD0010`
: It is a dynamic error if the match expression on p:viewport matches an attribute or a namespace node.

  See: p:viewport

`err:XD0012`
: It is a dynamic error if any attempt is made to dereference a URI where the scheme of the URI reference is not supported.

  See: Common errors

`err:XD0015`
: It is a dynamic error if a QName is specified and it cannot be resolved with the in-scope namespace declarations.

  See: System Properties, Step Available

`err:XD0016`
: It is a dynamic error if the select expression on a p:input or p:with-input returns attribute nodes or function items.

  See: p:with-input

`err:XD0017`
: It is a dynamic error if the running pipeline attempts to invoke an external step which the processor does not know how to perform.

  See: Extension Steps, Declaring external steps

`err:XD0019`
: It is a dynamic error if an option declares a list of acceptable values and an attempt is made to specify a value that is not a member of that list.

  See: p:option

`err:XD0020`
: It is a dynamic error if the combination of serialization options specified or defaulted is not allowed.

  See: Serialization parameters

`err:XD0021`
: It is a dynamic error for a pipeline to attempt to access a resource for which it has insufficient privileges or perform a step which is forbidden.

  See: Security Considerations

`err:XD0022`
: It is a dynamic error if a processor that does not support PSVI annotations attempts to invoke a step which asserts that they are required.

  See: PSVIs in XProc

`err:XD0028`
: It is a dynamic error if any attribute value does not satisfy the type required for that attribute.

  See: Common errors

`err:XD0030`
: It is a dynamic error if a step is unable or incapable of performing its function.

  See: Common errors

`err:XD0036`
: It is a dynamic error if the supplied or defaulted value of a variable or option cannot be converted to the required type.

  See: Variable and option types

`err:XD0038`
: It is a dynamic error if an input document arrives on a port and it does not match the allowed content types.

  See: Specifying content types

`err:XD0039`
: It is a dynamic error if the encoding attribute is present and content type value specifies a character set that is not supported by the implementation.

  See: p:inline

`err:XD0040`
: It is a dynamic error if the body is not correctly encoded per the value of the encoding attribute.

  See: p:inline

`err:XD0042`
: It is a dynamic error if a document arrives on an output port whose content type is not accepted by the output port specification.

  See: p:output

`err:XD0050`
: It is a dynamic error if the XPath expression in a value template can not be evaluated.

  See: Value Templates

`err:XD0051`
: It is a dynamic error if the XPath expression in an AVT or TVT evaluates to something to other than a sequence containing atomic values or nodes.

  See: Value Templates

`err:XD0052`
: It is a dynamic error if the XPath expression in a TVT evaluates to an attribute and either the parent is not an element or the attribute has a preceding node that it not an attribute.

  See: Text Value Templates

`err:XD0053`
: It is a dynamic error if a step runs longer than its timeout value.

  See: Controlling long running steps

`err:XD0054`
: It is a dynamic error if an encoding is specified and the content type is an XML media type or an HTML media type.

  See: p:inline

`err:XD0055`
It is a dynamic error if the content type value specifies a character set and the encoding attribute is absent.

See: p:inline

`err:XD0056`
It is a dynamic error if an encoding is specified and the content of the p:inline contains any XML markup.

See: p:inline

`err:XD0057`
It is a dynamic error if the text content does not conform to the JSON grammar.

See: Inline JSON content

`err:XD0061`
It is a dynamic error if $key is of type xs:string and cannot be converted into a xs:QName.

See: Document property, Special rules for casting QNames

`err:XD0062`
It is a dynamic error if the document-properties map contains a content-type key and that key has a value that differs from the statically determined content type.

See: p:inline

`err:XD0063`
It is a dynamic error if the p:inline contains any XML markup and has a content type that is not an XML media type or an HTML media type.

See: p:inline

`err:XD0064`
It is a dynamic error if the base URI is not both absolute and valid according to .

See: p:inline, p:document

`err:XD0065`
It is a dynamic error to refer to the context item, size, or position in a value template if a sequence of documents appears on the default readable port.

See: Value Templates, p:variable, p:with-option

`err:XD0068`
It is a dynamic error if the supplied value is not an instance of xs:QName, xs:anyAtomicType, xs:string or a type derived from xs:string.

See: Special rules for casting QNames

`err:XD0069`
It is a dynamic error if the string value contains a colon and the designated prefix is not declared in the in-scope namespaces.

See: Special rules for casting QNames

`err:XD0070`
It is a dynamic error if a value is assigned to the serialization document property that cannot be converted into map(xs:QName, item()*) according to the rules in Implicit Casting.

See: Document Properties

`err:XD0072`
It is a dynamic error if a document appearing on the input port of p:viewport is neither an XML document nor an HTML document.

See: p:viewport

`err:XD0073`
It is a dynamic error if the document returned by applying the subpipeline to the matched node is not an XML document, an HTML document, or a text document.

See: p:viewport

`err:XD0074`
It is a dynamic error if no absolute base URI is supplied to p:urify and none can be inferred from the current working directory.

See: Analysis

`err:XD0075`
It is a dynamic error if the relative path has a drive letter and the base URI has a different drive letter or does not have a drive letter.

See: Analysis

`err:XD0076`
It is a dynamic error if the relative path has a drive letter and the base URI has an authority or if the relative path has an authority and the base URI has a drive letter.

See: Analysis

`err:XD0077`
It is a dynamic error if the relative path has a scheme that differs from the scheme of the base URI.

See: Analysis

`err:XD0079`
It is a dynamic error if a supplied content-type is not a valid media type of the form "type/subtype+ext" or "type/subtype".

See: Specifying content types

`err:XD0080`
It is a dynamic error if the basedir has a non-hierarchical scheme.

See: Analysis

F.3. Step Errors

The following *dynamic errors* can be raised by steps in this specification:

Step Errors

`err:XC0023`
It is a dynamic error if a select expression or selection pattern returns a node type that is not allowed by the step.

See: Common errors

G. Guidance on Namespace Fixup (Non-Normative)

An XProc processor may find it necessary to add missing namespace declarations to ensure that a document can be serialized. While this process is implementation defined, the purpose of this appendix is to provide guidance as to what an implementation might do to either prevent such situations or fix them as before serialization.

When a namespace binding is generated, the prefix associated with the QName of the element or attribute in question should be used. From an Infoset perspective, this is accomplished by setting the `[prefix]` on the element or attribute. Then when an implementation needs to add a namespace binding, it can reuse that prefix if possible. If reusing the prefix is not possible, the implementation must generate a new prefix that is unique to the in-scope namespace of the element or owner element of the attribute.

An implementation can avoid namespace fixup by making sure that the standard step library does not output documents that require fixup. The following list contains suggestions as to how to accomplish this within the steps:

1. Any step that outputs an element in the step vocabulary namespace `http://www.w3.org/ns/xproc-step` must ensure that namespace is declared. An implementation should generate a namespace binding using the prefix "`c`".

2. When attributes are added by `p:add-attribute` or `p:set-attributes`, the step must ensure the namespace of the attributes added are declared. If the prefix used by the QName is not in the in-scope namespaces of the element on which the attribute was added, the step must add a namespace declaration of the prefix to the in-scope namespaces. If the prefix is amongst the in-scope namespace and is not bound to the same namespace name, a new prefix and namespace binding must be added. When a new prefix is generated, the prefix associated with the attribute should be changed to reflect that generated prefix value.

3. When an element is renamed by `p:rename`, the step must ensure the namespace of the element is declared. If the prefix used by the QName is not in the in-scope namespaces of the element being renamed, the step must add a namespace declaration of the prefix to the in-scope namespaces. If the prefix is amongst the in-scope namespace and is not bound to the same namespace name, a new prefix

and namespace binding must be added. When a new prefix is generated, the prefix associated with the element should be changed to reflect that generated prefix value.

If the element does not have a namespace name and there is a default namespace, the default namespace must be undeclared. For each of the child elements, the original default namespace declaration must be preserved by adding a default namespace declaration unless the child element has a different default namespace.

4. When an attribute is renamed by `p:rename`, the step must ensure the namespace of the renamed attribute is declared. If the prefix used by the QName is not in the in-scope namespaces of the element on which the attribute was added, the step must add a namespace declaration of the prefix to the in-scope namespaces. If the prefix is amongst the in-scope namespace and is not bound to the same namespace name, a new prefix and namespace binding must be added. When a new prefix is generated, the prefix associated with the attribute should be changed to reflect that generated prefix value.

5. When an element wraps content via `p:wrap`, there may be in-scope namespaces coming from ancestor elements of the new wrapper element. The step must ensure the namespace of the element is declared properly. By default, the wrapper element will inherit the in-scope namespaces of the parent element if one exists. As such, there may be a existing namespace declaration or default namespace.

   If the prefix used by the QName is not in the in-scope namespaces of the wrapper element, the step must add a namespace declaration of the prefix to the in-scope namespaces. If the prefix is amongst the in-scope namespace and is not bound to the same namespace name, a new prefix and namespace binding must be added. When a new prefix is generated, the prefix associated with the wrapper element should be changed to reflect that generated prefix value.

   If the element does not have a namespace name and there is a default namespace, the default namespace must be undeclared. For each of the child elements, the original default namespace declaration must be preserved by adding a default namespace declaration unless the child element has a different default namespace.

6. When the wrapper element is added for `p:wrap-sequence` or `p:pack`, the prefix used by the QName must be added to the in-scope namespaces.

7. When a element is removed via `p:unwrap`, an in-scope namespaces that are declared on the element must be copied to any child element except when the child element declares the same prefix or declares a new default namespace.

8. In the output from `p:xslt`, if an element was generated from the xsl:element or an attribute from xsl:attribute, the step must guarantee that an namespace declaration exists for the namespace name used. Depending on the XSLT implementation, the namespace declaration for the namespace name of the element or attribute may not be declared. It may also be the case that the original prefix is available. If the original prefix is available, the step should attempt to re-use that prefix. Otherwise, it must generate a prefix for a namespace binding and change the prefix associated the element or attribute.

H. Handling Circular and Re-entrant Library Imports (Non-Normative)

When handling imports, an implementation needs to be able to detect the following situations, and distinguish them from cases where multiple import chains produce genuinely conflicting step definitions:

1. Circular imports: A imports B, B imports A.

2. Re-entrant imports: A imports B and C, B imports D, C imports D.

One way to achieve this is as follows:

[Definition: The *step type exports* of an XProc element, against the background of a set of URIs of resources already visited (call this set *Visited*), are defined by cases.]

The step type exports of an XProc element are as follows:

p:declare-step
: A singleton bag containing the `type` of the element

p:library
: The *bag-merger* of the *step type exports* of all the element's children

p:import
: Let *RU* be the actual resolved URI of the resource identified by the `href` of the element. If *RU* is a member of *Visited*, then an empty bag, otherwise update *Visited* by adding *RU* to it, and return the *step type exports* of the document element of the retrieved representation

all other elements
: An empty bag

The changes to *Visited* mandated by the `p:import` case above are persistent, not scoped. That is, not only the recursive processing of the imported resource but also subsequent processing of siblings and ancestors must be against the background of the updated value. In practice this means either using a side-effected global variable, or not only passing *Visited* as an argument to any recursive or iterative processing, but also *returning* its updated value for subsequent use, along with the bag of step types.

Given a pipeline library document with actual resolved URI *DU*, it is a *static error* (`err:XS0036`) if the *step type exports* of the document element of the retrieved representation, against the background of a singleton set containing *DU* as the initial *Visited* set, contains any duplicates.

Given a top-level pipeline document with actual resolved URI *DU*, it is a *static error* (`err:XS0036`) if the *bag-merger* of the *step type exports* of the document element of the retrieved representation with the *step type exports* of its children, against the background of a singleton set containing *DU* as the initial *Visited* set, contains any duplicates.

Given a non-top-level `p:declare-step` element, it is a *static error* (`err:XS0036`) if the *bag-merger* of the *step type exports* of its parent with the *step type exports* of its children, against the background of a copy of the *Visited* set of its parent as the initial *Visited* set, contains any duplicates.

The phrase "a copy of the *Visited* set" in the preceding paragraph is meant to indicate that checking of non-top-level `p:declare-step` elements does *not* have a persistent impact on the checking of its parent. The contrast is that whereas changes to *Visited* pass both up *and* down through `p:import`, they pass only *down* through *p:declare-step*.

[Definition: The *bag-merger* of two or more bags (where a bag is an unordered list or, equivalently, something like a set except that it may contain duplicates) is a bag constructed by starting with an empty bag and adding each member of each of the input bags in turn to it. It follows that the cardinality of the result is the sum of the cardinality of all the input bags.]

I. Sequential steps, parallelism, and side-effects

XProc imposes as few constraints on the order in which steps must be evaluated as possible and almost no constraints on parallel execution.

In the simple, and we believe overwhelmingly common case, inputs flow into the pipeline, through the pipeline from one step to the next, and results are produced at the end. The order of the steps is constrained by the input/output connections between them. Implementations are free to execute them in a purely sequential fashion or in parallel, as they see fit. The results are the same in either case.

This is not true for pipelines which rely on side effects, such as the state of the filesystem or the state of the web. Consider the following pipeline:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                version="3.0">
<p:input port="source"/>
<p:output port="result"/>

  <p:xslt name="generate-stylesheet">
    <p:with-input port="source" href="someURI"/>
    <p:with-input port="stylesheet" href="someOtherURI"/>
  </p:xslt>

  <p:store name="save-xslt" href="gen-style.xsl"/>

  <p:xslt name="style">
    <p:with-input port="source">
      <p:pipe step="main" port="source"/>
    </p:with-input>
    <p:with-input port="stylesheet" href="gen-style.xsl"/>
  </p:xslt>
</p:declare-step>
```

There's no guarantee that "style" step will execute after the "save-xslt" step. In this case, the solution is straightforward. Even if you need the saved stylesheet, you don't need to rely on it in your pipeline:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                name="main" version="3.0">
<p:input port="source"/>
<p:output port="result"/>

  <p:xslt name="generate-stylesheet">
    <p:with-input port="source" href="someURI"/>
    <p:with-input port="stylesheet" href="someOtherURI"/>
  </p:xslt>

  <p:store name="save-xslt" href="gen-style.xsl"/>

  <p:xslt name="style">
    <p:with-input port="source">
      <p:pipe step="main" port="source"/>
    </p:with-input>
    <p:with-input port="stylesheet">
      <p:pipe step="generate-stylesheet" port="result"/>
    </p:with-input>
  </p:xslt>
</p:declare-step>
```

Now the result is independent of the implementation strategy.

Implementations are free to invent additional control structures using `p:pipeinfo` and extension attributes to provide greater control over parallelism in their implementations.

J. The `application/xproc+xml` media type

This appendix registers a new MIME media type, "`application/xproc+xml`".

J.1. Registration of MIME media type application/xproc+xml

MIME media type name:
`application`

MIME subtype name:
`xproc+xml`

Required parameters:
None.

Optional parameters:

`charset`
This parameter has identical semantics to the `charset` parameter of the `application/xml` media type as specified in [RFC 3023] or its successors.

Encoding considerations:
By virtue of XProc content being XML, it has the same considerations when sent as "`application/xproc+xml`" as does XML. See [RFC 3023], Section 3.2.

Security considerations:
Several XProc elements may refer to arbitrary URIs. In this case, the security issues of [RFC 2396], section 7, should be considered.

In addition, because of the extensibility features of XProc, it is possible that "application/xproc+xml" may describe content that has security implications beyond those described here. However, only in the case where the processor recognizes and processes the additional content, or where further processing of that content is dispatched to other processors, would security issues potentially arise. And in that case, they would fall outside the domain of this registration document.

Interoperability considerations:
This specification describes processing semantics that dictate behavior that must be followed when dealing with, among other things, unrecognized elements.

Because XProc is extensible, conformant "application/xproc+xml" processors can expect that content received is well-formed XML, but it cannot be guaranteed

that the content is valid XProc or that the processor will recognize all of the elements and attributes in the document.

Published specification:
This media type registration is for XProc documents as described by this specification which is located at http://www.w3.org/TR/xproc/.

Applications which use this media type:
There is no experimental, vendor specific, or personal tree predecessor to "`application/xproc+xml`", reflecting the fact that no applications currently recognize it. This new type is being registered in order to allow for the deployment of XProc on the World Wide Web, as a first class XML application.

Additional information:

Magic number(s):
There is no single initial octet sequence that is always present in XProc documents.

File extension(s):
XProc documents are most often identified with the extension "`.xpl`".

Macintosh File Type Code(s):
TEXT

Person & email address to contact for further information:
Norman Walsh, <Norman.Walsh@MarkLogic.com>.

Intended usage:
COMMON

Author/Change controller:
The XProc specification is a work product of the World Wide Web Consortium's XML Processing Model Working Group. The W3C has change control over these specifications.

J.2. Fragment Identifiers

For documents labeled as "`application/xproc+xml`", the fragment identifier notation is exactly that for "`application/xml`", as specified in [RFC 3023] or its successors.

K. Ancillary files

This specification includes by reference a number of ancillary files.

xproc30.rnc, **xproc30.rng**
: A RELAX NG Schema for XProc 3.0 pipelines, in compact or XML form.

xproc10.rnc, **xproc10.rng**
: A RELAX NG Schema for XProc 1.0 pipelines, in compact or XML form.

xproc.rnc, **xproc.rng**
: A RELAX NG Schema for XProc pipelines, in compact or XML form. It will validate either XProc 1.0 pipelines or XProc 3.0 pipelines, depending on the value of the version attribute.

  In order to use this schema, you must also download the 1.0 and 3.0 schemas; they are included into this one.

library.xpl
: An XProc pipeline library that declares all of the standard built-in steps.

L. Credits

This document is derived from XProc: An XML Pipeline Language published by the W3C. It was developed by the *XML Processing Model Working Group* and edited by Norman Walsh, Alex Miłowski, and Henry Thompson.

The editors of this specification extend their gratitude to everyone who contributed to this document and all of the versions that came before it.

M. Change Log

This list contains the non-editorial changes made after the August 2020 "last call" draft:

- The `depends` attribute is forbidden on `p:when`, `p:otherwise`, `p:catch`, and `p:finally`.

- Processors may remove any implicit connections that they can determine statically will never be used (issue 995).
- Expanded and clarified the rules for implicit casting (issues 1001 and 1012).
- The semantics of the `p:urify` function were extensively redrafted and clarified.
- Text value templates are never expanded in the descendants of `p:inline` elements that specify an `encoding`.
- Clarified the semantics of `[p:]use-when` to address potential deadlock situations that can arise if two or more expressions depend on each other.
- Clarified that `p:step-available` cannot refer to the step currently being declared (issue 1057).
- A number of error codes have been clarified and new error codes have been added.

This list contains the non-editorial changes made after the December 2019 "last call" draft:

- The `visibility` attribute of `p:variable` was removed.
- The description of the `select` attribute of `p:option` no longer mentions static variables.
- Error XD0079 added for defective content-types. (Was XS0070 or XS00130).

This list contains the non-editorial changes made after the February 2019 "last call" draft:

- The `p:document-properties-document()` function was removed
- The semantics of `p:if` have been changed. If the test expression is false, `p:if` behaves roughly like an identity step. Previously it produced no outputs.
- It is no longer a static error (XS0093), if `p:option` or `p:variable` have an attribute `visibility` and are not children of a `p:library`.

- The semantics of `p:choose` have been changed. The default sub-pipeline for a missing `p:otherwise` is a `p:identity` step (with the additional feature that it isn't an error if there's no default readable port). A primary output port on the `p:when` branches for this is required.
- The way the context item for XPath expressions is provided has been changed. It is now provided if and only if the connection delivers exactly one document, otherwise the context item is undefined. A new error (XD0001) is introduced. It is raised if an XPath expression makes use of the context item, but the context item is undefined. Two dynamic errors (XD0005 and XD0008) were removed.
- Content type shortcuts and the notion of forbidden content types have been added. See Section 3.4, "Specifying content types".
- Introduction of the `serialization` document property. See Section 3.1, "Document Properties".
- The semantics pf `p:viewport` have been changed. HTML documents are now allowed as input and `match` may now match every node type except attributes and namespace nodes.
- Static `p:variable`s have been removed; the semantics of static `p:option`s have been updated and clarified.
- Clarified that the `base-uri` property must always be a legal URI per [RFC 3986].
- Expanded the range of media types that may be considered "text" documents; opened the possibility for implementations to extend the list.
- Clarified that leading and trailing whitespace within a `p:inline` is not discarded.
- Identified the `error` port as the primary input port in `p:catch`.
- Clarified that `p:finally` **must not** declare a primary output port.
- Clarified which functions are available during static analysis.
- Clarified that atomic values are considered JSON documents.
- Made `href` required on `c:entry`.

- Clarified how sequences of XDM values (for example, from a `p:xslt` step) are converted into documents.
- Updated the description of the `select` attribute for `p:input` so that it is in line with the more recent changes that have been applied to `p:with-input`.
- Changed description of `p:viewport` stating that the base URI of every matched node is the document's base URI, not just for document and element nodes.
- Changed the default value for serialization property `omit-xml-declaration` from `true` to `false` in section “Serialization method” . Removed remark about default settings for this parameter from section “Minimal conformance”.
- Clarified the conditions under which steps may produce PSVI annotations.
- Added a `cause` attribute to the error vocabulary for recording the error codes of underlying errors.